Teljesértékű tankönyv és forrásmű

minden komoly és leendő Delphi programozónak

Marco Cantù

Delphi 3

Mesteri Szinten

I. KÖTET

SYBEX®

KISKAPU

Kiskapu Kiadó

Delphi 3

Mesteri Szinten

I. KÖTET

KISKAPU KIADÓ

A kiadvány a következő angol eredeti alapján készült:
Marco Cantù: Mastering Delphi 3 second edition

Copyright © 1997 Sybex Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

Associate Publisher: *Gary Masters*
Acquisitions Manager: *Krisine Plachy*
Desktop Publisher: *Franz Baumhackl*
Technical Editor: *Juancarlo Añez*
Editor: *James A. Compton*
Graphic Illustrator: *Inbar Berman*
Production Coordinator: *Alexa Riggs*
Indexer: *Nancy Guenther*
Proofreaders: *Jennifer Matzger, Theresa Gonzalez, Michael Tom*
Acquisitions & Developmental Editor: *Peter Kuhns*

SYBEX is a registered trademark of SYBEX Inc.
Mastering is a trademark of SYBEX Inc.

Microsoft Windows, Windows 95 and Windows NT are registered trademarks of Microsoft Corporation.
Netscape Communications, the Netscape Communications logo, Netscape, and Netscape Navigator are trademarks of Netscape Communications Corporation.

A szerző és a kiadó a lehető legnagyobb körültekintéssel járt el e kiadvány elkészítésekor, a könyv tartalmát a szoftver végső kiadásához igazították, ahol ez lehetséges volt. A kézirat bizonyos részei a szóban forgó szoftverek úgynevezett "prerelease" verzióin alapul, mely programokat azok gyártói biztosították a kiadvány elkészítéséhez. Sem a szerző, sem a kiadó nem vállal semminemű felelősséget vagy garanciát a könyv tartalmával, teljességével kapcsolatban. Sem a szerző, sem a kiadó nem vonható felelősségre bármilyen baleset vagy káresemény miatt, mely közvetve vagy közvetlenül kapcsolatba hozható e kiadvánnyal.

Magyar változat © 1998 Kiskapu Kft. Minden jog fenntartva!
Hungarian edition © 1998 Kiskapu Kft. All rights reserved!

Sorozatszerkesztő: *Szy György*
Műszaki szerkesztő: *Troják Miklós*
Fordításvezető: *Tanács Lajos*
Lektorok: *Troják Miklós, Szy György*
Fordítók: *Kisteleki Róbert, Nagy Imre, Nagy Rita, Nagy Szabolcs, Németh Gábor, Mihalicza József, Török András*

Felelős kiadó a Kiskapu Kft. ügyvezető igazgatója
© 1998 Kiskapu Kft.
1081 Budapest Népszínház u. 29.
Tel: (+36-1) 303-9119 Fax: (+36-1) 303-1619
http://www.kiskapu.hu/ e-mail: kiskapu@kiskapu.hu

ISBN: 963 03 5348 2
Összesített ISBN: 963 03 5347 4 ö

Feleségemnek, Lellának

Köszönetnyilvánítás

Sok idő eltelt azóta, hogy elkezdtem összeállítani e könyv első kiadását. Azóta sok minden történt, sok emberrel kapcsolatba kerültem mind munkában, mind a magánéletben, és nem könnyen találom a szavakat, hogy megköszönjem mindannyiuknak a segítséget.

Minden bizonnyal első sorban a Borland cég programozóinak és menedzsereinek kell megköszönjem, mivel ők hozták létre ezt a kitűnő programot. Különösen köszönöm Zack Urlocker és David Intersimone segítségét, akik már hosszú évek óta segítenek nekem, de ugyanez a köszönet illeti a Borland Delphi csapatot (nevüket láthatjuk, ha a Delphi About ablakában az Alt gomb nyomva tartása mellet begépeljük: **TEAM** vagy **DEVELOPERS**). A könyv sokat köszönhet Nan Borresonnak, aki először ajánlotta nekem, hogy írjak egy Delphi könyvet, és sok tanáccsal segítette annak megjelenését.

A következő köszönetet a Sybex kiadó szerkesztőinek címzem. Sokakat nem is ismerek személyesen azok közül, akik részt vállaltak e könyv megjelenésében, de mindannyiuknak köszönöm. Különösen köszönöm Gary Masters, Barbara Gordon, Jim Compton, Peter Kuhns, Kristine Plachy, Francz Baumhackl, Inbar Berman, alexa Riggs, Theresa Gonzalez, Jennifer Metzger, Michael Tom és Molly Sharp segítségét. Ezen kívül köszönet az előző kiadás elkészítésében segítőknek is: John Read, Judy Jugarjian, Heather O'Connor, Thom Dyson, Jane Reh, Mailyn Smith, Dusty Bernard, Valerie Potter, Shelby Zimmerman és Michelle Nance.

Juancarlo Añez, Tim Gooch, Alain Tadros és Danny Thorpe nézte át a három kiadást, és új ötletekkel és példákkal segítették a könyv fejlődését. Sok olyan lehetőségre hívták fel a figyelmem, melyet nem ismertem, és sok hiba kijavítását köszönhetem nekik. A könyvem szakmai színvonalát leginkább szakmai lektoraim garantálják, ezért külön köszönet nekik.

Nagyon sok emberrel vitattam meg Delphi programozással kapcsolatos kérdéseket a fent említetteken kívül a világ sok pontján; USA, Olaszország, Franciaország, Anglia, Szingapúr, Hollandia, hogy csak néhány országot soroljak fel. Bruce Eckel, jó barátom, és több C++ és Java könyv írója segített jobban megérteni az OOP rejtelmeit, és tiszta szívből köszönöm neki az együtt töltött időt.

Sok újságíró is segített nekem, mint például Chris Frizelle a The Delphi Magazine-tól, JD Hildebrand a Windows Tech Journal-tól, Bob Arnson, Kevin Weeks (aki kikiáltotta az első kiadást a "Manhattan telefonkönyvéhez legjobban hasonlító könyv" díj nyertesének), Jerry Coffrey a Delphi Informant-tól, Giorgio Panzeri a PcProfessionale-tól és Alessandro Pedone a Computer Programmingtól.

Sokan vannak még, akiknek sokat köszönhetek: Norm McIntosh, aki többször is elszállásolt San Francisco-i házában, Andrea Provaglio, egy OOP specialista, akivel többször találkoztam az államokban, mint Olaszországban, Steve Tendon, aki az első kéziratot teljes egészében átrágta, Bob Swart vagy másik nevén Dr. Bob, Johanna and Phil az UK Delphi Developer's Group tagjai, Stefano Maruzzi, aki a Windows programozással kapcsolatban mutatott sok újat nekem, a felesége, Antonella, Foo Say How, Szingapúrból, Giovanni Librando, aki bemutatta nekem a Visual Basic programnyelvet (és így rávilágított a Delphi előnyeire), Glenn Field, John Howe, Mike Orriss, a TeamB tagjai a Compuserve Delphi fórumáról, Marco Miotti, aki most a Microsoft csapatában dolgozik, valamint Ernesto Franchini, kis csapatával az ISS Borland-on belül. Emellett nagy köszönet az összes Delhpi szemináriumhallgatómnak, és Jam Sessionnak.

Ezen emberek mellett nagyon sokan vannak, akik segítettek nekem, ha nem is kifejezetten szakmai kérdésekben. Elsősorban feleségem, Lella, aki minden lehetséges módon támogatott. Nagyon sok tervet – főleg a szabadságokkal, üdülésekkel kapcsolatosakat – módosítania kellett. Néha rendkívül bosszantó ilyen időpontokat megváltoztatni, ő mégis végig kitartott mellettem, és amikor látta, hogy nem úgy alakulnak a dogok, ő parancsolt pihenőt.

Sok barátomnak szeretném megköszönni, hogy lehetőséget adtak a kikapcsolódásra azzal, hogy meghívtak minket egy-egy délutánra magukhoz vagy egy filmre. Még egy közösen elfogyasztott pizza, vagy akár egy kis politizálás is üdítőleg hatott rám. A barátaim listája – szerencsére – elég hosszú, de külön kiemelném a következőket: Sandro és Monica, Stefano és Enela, Marco és Laura (és most már Matteo), Chiara, Luca és Elena, Chuara és Danielle, valamint Laura. Szüleink, tesvéreink és az ő családjuk is sokat segített. Nagy öröm volt szabadidőnkben találkozni velük és a hat unkaöccsel, Matteoval, Anderával, Giacomoval, Stefanoval, a másik Andreával, és Pietroval. A velük töltött játékidő segített kikapcsolódni.

És végül szeretném megköszönni mindazoknak – sokuknak ismeretlenül is – akik élvezik az életet, és segítenek egy jobb világ kialakításában. Hogy sosem adom fel a jövőbe vetett bizalmamat, nagyrészt nekik is köszönhető.

Előszó a Magyar Kiadáshoz

Először is a talán legelőször felmerülő kérdésre hadd válaszoljak. Az angol eredeti ezerötszáz oldalas, és egy CD melléklete is van. Ez a kötet mindössze nyolcszáz oldal környékén mozog és nincs melléklet hozzá. Hogyan lehet ez? Nos, a könyvet – főleg nyomdatechnikai okokból – két kötetben adjuk ki. Az első kötet tartalmazza az eredeti mű első két részét, míg a második kötet a könyv harmadik és negyedik részét. A CD is a második kötet melléklete.

Nagyon sok kérdést vet fel egy ilyen könyv hazai kiadása. Igyekeztünk, hogy minden igénynek megfeleljünk. Mégis, mielőtt szó érné a ház elejét, hadd írjak le néhány dolgot, melyet a könyv fordítása közben szem előtt kívántunk tartani.

A szakszavaknál nagy gondban voltunk. Sok olyan szó szerepel a könyvben, melyre nehezen, nagy viták után találtunk magyar megfelelőt. Sok angol kifejezés helyett, melynek már van használatban magyar változata (pl. Toolbar – Eszköztár, vagy CheckBox – jelölő négyzet), próbáltuk a fordítást használni. Ez néha elég fájdalmas, hisz a ComboBox-ot nagyon sokan használják, míg a KombiPanel elnevezés aránylag új, még a szakmában nem annyira elterjedt. Sok kifejezést viszont nem fordítottunk le, vagy azért, mert nem volt értelme lefordítani (például ilyen a form, melyet sem ablaknak nem szabad fordítani – hisz annál több –, sem űrlapnak – hisz annál pedig kevesebb), vagy azért, mert nem akartunk rá egy "kényszerített" magyar kifejezést ráhúzni (minek neveztük volna például a CoolBar-t).

Nagy vita volt a kiadón belül a mintaprogramok kérdése. Pontosabban az a kérdés, hogy a forráskódot lefordítsuk-e. Ez ügyben a következőre jutottunk: A programkódot úgy hagyjuk, ahogy van. Ez nem (csak) lustaságunk jele, hanem egyszerűen a Delphi nyelvezetének követése. Véleményem szerint ha valaki komoly (esetleg nemzetközi) programot ír, azt angolul tegye, hisz maga a programnyelv is angol, és ennek a szakmának is ez a nyelve. Nem beszélve arról az érvről, hogy így a könyv előállítási költsége alacsonyabb, tehát a vásárlók pénztárcáját is megóvtuk.

Kérem Önt, kedves Olvasó, hogy bármilyen véleménye, kritikája van a könyvvel kapcsolatban, írja meg nekünk, akár a könyv hazai kiadásának honlapján, a http://www.kiskapu.hu/konyvek/delphi/ címen, akár e-mail-ben a kiskapu@kiskapu.hu postaládába, akár a Kiadó impresszumban található postacímére.

Jó olvasást és jó programozást kívánva,

A sorozatszerkesztő,

Szy György

Tartalomjegyzék

I. RÉSZ 1

A Form egy ablak (és egy ActiveX) 5
Első formunk elkészítése 5
Cím beállítása 6
A form elmentése 8
Komponensek használata 8
Tulajdonságok megváltoztatása 9
Reagálás az eseményekre 12
A program fordítása és futtatása 16
Tulajdonságok futásidejű megváltoztatása 18
Kód hozzáadása a programhoz 20
Egy kétirányú eszköz 22
A forráskód 22
A form szöveges leírása 24
A projekt-állomány 27
ActiveForm készítése 28
Komponens-sablonok használata 29
ActiveX könyvtár és ActiveForm 29
A HTML oldal 31
Hogyan tovább? 33

A Delphi-környezet jellemzői 37
A Delphi különböző verziói 37
Segítség kérése 38
Delphi menük és parancsok 40
A File menü 40
Az Edit menü 42
Komponensek másolása és beillesztése 42
További szerkesztő parancsok 45
A Search menü 45
A View menü 48
A Project menü 49
A Run menü 50
A Component menü 51
Komponens sablonok 51

A Database menü 51
A Tools menü 52
A Help menü 53
A Delphi eszköztára 53
A lokális menük 54
A Form Designer használata 54
A komponenspaletta 57
Az Object Inspector 58
Az Alignment paletta 59
Programkód írása a szerkesztőben 60
A szerkesztő könyvjelzőinek használata 60
A Delphi 3 Code Insight szolgáltatásai 62
Kódkiegészítés 62
Kódsablonok 63
Kódparaméter 64
Projektek menedzselése 65
A Project Manager 65
Projektopciók beállítása 66
Egy projekt lefordítása 67
Egy lefordított program vizsgálata 68
A beépített hibakereső 68
Az Object Browser 69
Egyéb Delphi eszközök 69
A rendszer fájljai 70
Hogyan tovább? 75
Az Object Repository és a Delphi varázslók 79
Az Object Repository 79
A New oldal 81
Az ActiveX oldal 82
Az „aktuális projekt" oldal 83
A Forms oldal 83
A Dialogs oldal 84
A Data Modules oldal 85
A Projects oldal 85
Delphi varázslók 86
Az Adatbázis Form varázsló (Database Form Wizard) 87
Az Alkalmazás varázsló (Application Wizard) 89
A Párbeszédablak varázsló (Dialog Wizard) 91

Az Object Repository testreszabása 92
Új alkalmazás sablon felvétele *92*
Az üres projekt sablon 94
Új form sablonok hozzáadása 94
Az Object Repository opciói *95*
Új varázsló DLL-ek telepítése 96
Hogyan tovább? 97
A Pascal nyelv 101
Típusok, változók és konstansok 101
Változók *101*
Konstansok *103*
String erőforrás (resource string) konstansok *104*
Delphi adattípusok *104*
Sorszámozott típusok (ordinal types) *105*
A Range példaprogram *106*
Sorszámozott típusokon használható eljárások és rutinok *108*
Valós típusok *109*
Dátum és idő *110*
Különleges Windows-típusok *114*
Ablakkezelők (handle) *115*
Típusmegfeleltetés (casting) és típuskonverzió *116*
A Variant típus *118*
Variant típusok a Delphiben 120
A Variant lassú! 121
Felhasználó által meghatározott adattípusok *122*
A részintervallum típusok (subrange types) *124*
Felsorolt típusok *125*
Halmaz típusok *127*
Tömb típusok (array types) *129*
A rekord adattípus *131*
A mutató típus (pointer) *133*
Állománytípusok *135*
Stringek a Delphiben 136
A hagyományos Pascal stringek *136*
A Delphi hosszú stringjei *136*
StrRef: Példa a dinamikus string allokálásra *138*
C-stílusú karaktertömbök *140*
String átalakítások *141*
Stringátalakítási nehézségek *142*
Stringek formázása *144*
Programozási stílus 146
Megjegyzések *147*

A NAGY- és kisbetűk használata *148*
Közök (white space) *148*
A szép külalak *149*
A szintaktikai elemek kiemelése *150*
Kódsablonok használata *151*
Nyelvi kifejezések 152
Kifejezések és operátorok *152*
Operátorok és a precedencia szabály *153*
Unáris (egyoperandusú) operátorok 153
Multiplikatív és bitenkénti operátorok 153
Additív operátorok 153
Relációs és összehasonlító operátorok 154
Halmaz operátorok 154
Egyszerű és összetett kifejezések *154*
Elágazások 156
Az if utasítás *156*
A Case utasítás *158*
Ciklusok a Pascalban 158
A for ciklus *158*
A while és a repeat utasítások *159*
A with utasítás *162*
Eljárások és függvények 165
Cím szerinti paraméterátadás *166*
Konstans paraméterek *167*
Nyitott tömb paraméterek *167*
Változtatható típusú nyitott tömb paraméterek *168*
A Delphi alprogram-hívási lehetőségei 171
Mi is az a metódus? *172*
Elődeklarációk *172*
Külső deklarációk *174*
Eljárás-típusok *175*
Windows callback függvények *178*
Hogyan tovább? 180
Az Object Pascal, mint objektumorientált nyelv 183
Osztályok és objektumok 183
A Delphi objektum-hivatkozási modellje *186*
A TDate osztály felhasználása Delphiben *188*
Konstruktor készítése *190*
Objektumok a memóriában *192*
Az osztályok és az információ-rejtés 194
Private, protected, public *195*
Public és published *196*

Osztályok és egységek 197
Osztály interfész *201*
Egységek és érvényesség *202*
Egységek és érvényességi körök *203*
Változások elrejtése *204*
Egység a TDate osztály számára *206*
Egységek és programok *210*
Öröklés meglévő típusokból *211*
Öröklés és típus kompatibilitás *215*
Késői kötés és polimorfizmus *218*
Metódusok felülírása és újradefiniálása *221*
Virtuális és dinamikus metódusok *222*
Üzenetkezelők *223*
Absztrakt metódusok *224*
Futási idejű típus információ 228
Kivételkezelés 230
Az at kulcsszó *232*
Példa kivételek használatára *232*
Kivételek és a verem *236*
A finally blokk *237*
Hogyan tovább? 241
Haladó szintű Object Pascal 245
A Self kulcsszó 245
Komponensek dinamikus létrehozása 246
Osztálymetódusok és adatok 248
Egy osztály objektum-számlálóval 249
Metódus-mutatók 253
A számlálós példa fejlesztése 255
Objektumok egyszeri megsemmisítése 257
Osztályreferenciák 259
Objektumok létrehozása futási idő alatt osztályhivatkozások használatával 261
Interfész és többszörös öröklődés 263
Interfész deklarálása *264*
Interfész használata paraméterként *267*
Egy második interfész írása *268*
Mindkét interfész implementációja *270*
Tulajdonságok és események definiálása 273
Tulajdonságok hozzáadása a TDate osztályhoz *276*
Események a Delphiben 281

Az események tulajdonságok *281*
Egy esemény hozzáadása a TDate osztályhoz *282*
Egy TDate komponens készítése *284*
Hogyan tovább? 286

A Vizuális Komponenskönyvtár (Visual Component Library) **289**

A TObject osztály 289
Az osztályinformáció megjelenítése *292*
A VCL hierarchiája 293
A komponensek *295*
Az objektumok *296*
A kivételek *300*
A VCL osztályok használata *302*
Az általános VCL tulajdonságok 302
A Name (Név) tulajdonság *305*
Komponensek méretére és helyzetére vonatkozó tulajdonságok *308*
Aktivizálási és láthatósági tulajdonságok *308*
A testreszabható Tag tulajdonság *309*
A felhasználói interfész: színek és fontok *310*
Általános VCL metódusok 312
Az általános VCL események 315
A Delphi kollekcióinak használata 316
Objektum- és adatlisták használata *318*
A VCL forráskód tanulmányozása 320
Hogyan tovább? 321

II. RÉSZ 323

Alapvető komponensek **327**

A Windows saját komponensei 327
Gombra kattintás 328
A Buttons (gombok) példa *329*
Kattintás az egérrel 333
Színes szövegek a formon 334
A LabelCo példa *334*
A standard szín párbeszédablak *337*
Elemek áthúzása komponensek között 338
A Dragging példa kódja *340*
Felhasználói input fogadása 341
A bemeneti fókusz kezelése *342*
Egy általános OnEnter eseménykezelő *346*
Számok bevitele *347*

Kifinomult beviteli sémák 352
Egy egyszerű szerkesztő készítése 354
A Font párbeszédablak *354*
Egy komolyabb szerkesztő készítése *355*
Választások 357
A rádiógombok csoportosítása *358*
A Phrases1 példa *359*
Egy lista sok választási lehetőséggel 363
A Phrases2 példa formja *363*
Listapanelek kezelése *365*
A kiválasztott string eltávolítása a másik listapanelből *366*
Többszörös kiválasztás engedélyezése 370
A Phrases példa harmadik változata *372*
A jelölőnégyzet-lista (CheckListBox) komponens használata *375*
Sok lista, kevés hely 377
Egy adott határon belüli érték kiválasztása 379
A Scroll Color példa *379*
Hogyan tovább? 382
Menük készítése és kezelése 385
A főmenü felépítése 385
A menüpontok különböző szerepei *386*
Menü készítése a Menü Tervezővel *387*
A menü általános felépítése *388*
Gyorsbillentyűk, forróbillentyűk *388*
Előre definiált menüsablonok használata *389*
Menüparancsok feldolgozása 390
A Menü Tervező által generált kód *391*
A MenuOne példa forrása *393*
Menük módosítása futásidőben 395
Menüpontok változtatása futásidőben *395*
Menüpontok kikapcsolása és legördülő menük elrejtése *397*
Rádió menüpontok használata *399*
Menüpontok készítése dinamikusan *400*
Menük és menüpontok létrehozása dinamikusan 405
Rövid és hosszú menük *405*
Grafikus menüpontok 407
A pipa módosítása *407*
Kép menüpontok *411*
Tulajdonos által rajzolt menüpontok *415*
A rendszermenü testreszabása 419

Teljes menü készítése 422
A Fájl menü 423
Rövid kiértékelés 425
A Bekezdés menü 429
A Betűtípus menü 430
A Beállítások menü 433
Előugró menük 434
Automatikus helyi menü 435
Előugró menü módosítása aktiváláskor 436
Előugró menük kezelése programból 437
Hogyan tovább? 439
A formok mélyebb megismerése 443
Formok és ablakok 443
Átlapolt, felbukkanó és gyermekablakok 445
Az alkalmazás is ablak 446
Form-stílusok beállítása 447
Legfelül maradó form készítése 448
A villanás kiküszöbölése 449
Keretstílus 451
A keretstílus hatása 452
A Borders példa 453
A keret ikonjai 454
További ablakstílusok használata 456
A formok manuális méretezése 457
Formok automatikus méretezése 460
Méretezés és fontok 461
A form helyzetének és méretének állítása 461
A form kis- és teljes méretűvé alakítása 463
A form és kliensterületének mérete 465
A formok maximális és minimális mérete 466
Formok készítése automatikusan 468
A form bezárása 469
A billentyűzet-események figyelése 472
Egéresemények figyelése 475
Az egér gombjai 475
A Windows használata egér nélkül 476
Az egéresemények paraméterei 476
Kattintás és rajzolás: a Shapes1 példa 477
Rajzolás a formra 479
A rajzeszközök 480
Alakzatok rajzolása 481

Rajzolás és festés a Windows-ban 484
Egy egyszerű alakzat festése 485
Több alakzat festése 487
A Delphi kimeneti komponensei 491
Hogyan tovább? 491
Grafikus komponensek 495
A felhasználói felület grafikus fejlesztése 495
Bitmap a gombon 496
Egy autó a gombon 497
Változó kép a gombon 502
A kétállású gomb 502
Több kép egy bitmapben 504
A forgó földgömb 506
Bitmap lista, erőforrások használata, és a rajztábla 508
Az Animáció kontrol 514
Kép megjelenítő 517
Rajzolás a bittérképbe 520
Alakzatok rajzolása 521
Grafikus listák 524
Színek listájának kirajzolása 524
A könyv rövid áttekintése 528
A fejezetek struktúrája 532
Az Outline csomópontjai 536
Egy grafikus lista 539
Grafikus rácsok 542
A font rács 544
Aknák a rácsban 548
Színek választása 553
A TeeChart használata 554
Az első példa készítése 556
Adatok hozzáadása a grafikonhoz 558
Sorozatok dinamikus létrehozása 560
A Dátum–Idő kiválasztó kontrol 562
Hogyan tovább? 564
Eszköztár, CoolBar, státuszsor 567
Kontrolok csoportosítása paneleken 567
Eszközsor készítése panelből 568
Az első eszköztár 569
Eszköztár gombok ki és bekapcsolása 572
Tippek hozzáadása az eszköztárhoz 574

Tippek a formokon 576
Tippek testre szabása 576
Több tipp egy kontrolhoz 578
Szolgáltatások az eszköztáron 580
Combo box eszköztáron 580
Mozgatható eszköztár 582
A Windows 95-ös Toolbar (Eszköztár) kontrol 586
Egy remek eszköztár 590
Státuszsor készítése 593
Menü tippek a státuszsoron 594
Speed Button tippek a státuszsoron 596
Hogyan tovább? 597
Több form és párbeszédablak 601
Párbeszédablakok vagy formok? 601
Második form hozzáadása a programunkhoz 602
Modális és nem modális formok 605
Két form, két menü 607
Form menük egybeolvasztása 608
Párbeszédablakok készítése 611
Modális párbeszédablakok 611
Párbeszédablak bezárása 614
Nem modális párbeszédablak 615
Előre definiált párbeszédablakok használata 620
Általános Windows párbeszédablakok 620
Az Open Dialog komponens 620
A Font Dialog komponens 622
A Color Dialog komponens 624
A Find Dialog és Replace Dialog komponensek 624
Üzenetablakok sokasága 626
Bővíthető párbeszédablakok 630
Névjegy készítése 633
A rendszer Névjegyének használata 633
Saját rejtett képernyő készítése 634
Nyitóképernyő készítése 637
Vizuális form öröklés 642
Öröklés alapformból 643
Polimorf formok 646
Hogyan tovább? 650
Görgetés, többlapos formok 655
Amikor a formok túl nagyok 655

A form görgetése 656
A görgetést tesztelő példa 657
Automatikus görgetés 660
Kép görgetése 660
Görgetés és form koordináták 662
Jegyzettömbök készítése a Delphivel 664
PageControl és TabSheet komponensek 665
Notebook TabSet-tel 670
Jegyzettömb oldalának megváltoztatása 674
Füllel rendelkező jegyzettömbök 677
Jegyzettömb fülek nélkül és fülek jegyzettömb nélkül 678
Megjelenítés egy jegyzettömbben 679
Egy képkezelő program 680
Képnézegető fülekkel 683
Egy többoldalas eszköztár 684
Formfelosztási technikák 686
Elválasztás Splitterrel 686
Vízszintes elválasztás 689
Alternatív elválasztó komponensek 690
Elválasztás fejléccel 691
Elválasztás panelekkel 694
Elválasztó vonal rajzolása 697
Közvetlen egérkezelés a formban 701
Húzás egérrel 701
A húzás kódja 702
Hogyan tovább? 703
MDI alkalmazások készítése 707
MDI a Windowsban: technikai áttekintés 707
Keret és gyermekablakok a Delphiben 710
Az első MDI demo Delphiben 710
Egy teljes Window menü felépítése 713
Egy gyermekablak létrehozása 715
MDI alkalmazások különböző gyermekablakokkal 717
Egy pattogó alak hozzáadása 718
Az új gyermekform menüje 719
A főform átalakítása 720
Az MDI gyors elsajátítása 722
Hogyan tovább? 724

Adatbázis-kezelő alkalmazások készítése 727
Adatok, fájlok, adatbázisok, és táblák 727
Mi is egy tábla? 730
Műveletek végzése az adatbázison 731
A Delphi adatbázis komponensei 731
Táblák és lekérdezések 733
Az adatkészlet állapota 735
Más adatkezelő komponensek 736
A Delphi adatfüggő komponensei 736
Adatbázis-kezelő alkalmazások készítése manuálisan 738
Egy adatbázis-kezelő rács 738
A rács testreszabása 740
A tábla állapota 741
DBEdit vezérlők használata 742
Lekérdezések használata 744
Paraméterezett lekérdezések 748
A Database Form varázsló használata 750
Az adatmezők elérése 751
A mezőosztályok hierarchiája 753
Számított mező készítése 757
Mezők használata a tábla változtatásához 762
Rekordok keresése a táblában 762
A kereső metódusok 764
Goto metódusok 765
A Locate metódus 766
Egy tábla oszlopának szummázása 766
A tábla egy oszlopának szerkesztése 769
Az adatbázis különböző tábláinak használata 770
Az adatbázis és a tábla kiválasztása futásidőben 770
Egy táblaböngésző program 772
Egy mezőszerkesztő program 775
A táblaböngésző felhasználói felületének továbbfejlesztése 776
Tábla létrehozása 778
Táblák dinamikus létrehozása 779
A form és a kezdő programrészlet 781
Új tábla létrehozása 782
Egy létező tábla kiválasztása a megfelelő mezők segítségével 784
Rekordok hozzáadása és törlése 787
Többrekordú rács használata 788
Rácspanelek mozgatása 789

Kapcsolttáblás form készítése varázslóval 791
A kapcsolttáblás szerkezet használata lekérdezésekben ..793
Választás kombinált lista segítségével 793
Keresés a rácsban 795
Mi következik? 797
Haladó adatbázis-kezelés 801
Kapcsolat létesítése SQL szerverrel 801
Első InterBase alkalmazásunk 802
Kapcsolat távoli SQL szerverrel 804
InterBase szerver eszközök 804
Meglévő program átírása SQL szerverre 807
A tábla átmásolása 808
Az alkalmazás átírása 810
A Visual Query Builder használata 811
Átírás vagy fejlesztés? 814
Tábla-összekapcsolások a Visual Query Builderrel 814
Egy háromtáblás összekapcsolás 815
Egy többtáblás kapcsolat 817
Adatmodulok 819
Egy adatmodul elkészítése 820
Adatmodul több nézethez 821
Mezőtulajdonságok és kezdőértékek megadása 823
Egyszerű táblaszűrés 825
Egyéni táblaszűrés 827
Egyéni szűrés és a kliens/szerver fejlesztés 829
Az adatszótár 830
Az adatszótár és a mezőszerkesztő (Fields Editor) 831
Miből áll egy attribútumhalmaz? 832
Az adatszótár használata 833
Tranzakciókezelés 834
Egy egyszerű példa a tranzakciókezelésre 835
Egyéni adatbázis-belépés 837
Adatbázishibák kezelése 839
Hogyan tovább? 843

Bevezető

Az egész sok évvel ezelőtt kezdődött. Boldog C++ programozó voltam, amikor egy ismerősöm, Zack Urlocker megmutatott egy megjelenés előtt álló terméket, amelynek a neve Delphi volt. Már első látásra megtetszett annak nyelvezete, gazdag fejlesztői környezete, és használatának egyszerűsége. A Pascal volt az első komoly programozási nyelv, amivel az egyetemen megismerkedtem, nem is tartott sokáig felfrissíteni ezzel kapcsolatos ismereteimet, és igen hamar belejöttem az Object Pascal használatába is. Amint egy kicsit közelebbről is megismerkedtem a Delphi-vel, már sejtettem, hogy igen népszerű fejlesztői eszköz válik majd belőle, és amint azt valószínűleg tudjuk, igazam is lett.

Korábban a C és C++ nyelvek használatát oktattam, és erről a témáról írtam könyveket – ma szinte kizárólag a Delphi-vel foglalkozom. A Delphi népszerűségének és gyors elterjedésének köszönhetően e könyv korábbi kiadásai is óriási népszerűségnek örvendtek.

A Delphi-től a Delphi 3-ig

A Delphi megjelenésétől fogva kiváló választásnak bizonyult, ha hatékony programozási környezetet keresett az ember. Számomra form-központúsága, szupergyors fordítója, nagyszerű adatbázis-kezelése, a Windows működésébe való szerves illeszkedése, és komponenseken alapuló technológiája tartozik legvonzóbb tulajdonságai közé. Legfontosabb összetevője azonban mind közül az Object Pascal, ami az egész fejlesztői eszköz alapját képezi.

Az ezután megjelenő Delphi 2 elődjét is felülmúlta. A legfontosabb benne található újítások a többrekordú objektumok és a továbbfejlesztett adatbázis-rács, az OLE Automation támogatás és a variant típus bevezetése, teljes körű Windows 95 támogatás, a long string adattípus és a Visual Form Inheritance (vizuális formöröklődés) megjelenése voltak.

A Delphi 3 megjelenésekor teljesen lenyűgözte a programozókat. Ez a változat olyan sok új tulajdonsággal rendelkezik, hogy nehéz kiválasztani, melyik is a legjobb, így inkább összeállítottam egy fordított sorrendű, tízelemű listát, amely saját kedvenceimet tartalmazza:

10. A Code Insight technológia (Code Templates (kódsablonok), Code Completion (kódkiegészítés), Code Parameters (kódparaméterek) és Tool-Tip kifejezés kiértékelés), valamint a szerkesztő változásai.
9. DLL hibakeresési lehetőségek.
8. A komponenssablonok.

7. A megszokott vezérlők kiterjesztett változatai, mint például a Coolbar komponens.
6. A BDE új Access meghajtója, és az adatbázis-kapcsolatok rugalmasabbá tétele.
5. A TeeChart és a Decision Cube megjelenése, valamint a QuickReport új verziójának használata.
4. A hálózati és Internetes fejlesztések támogatása.
3. A komponenscsomagokat használó techniológia.
2. Az ActiveForm-ok.
1. Az interface-ek és a COM támogatás.

Hogyan sajátíthatjuk el a Delphi használatát? (a könyv segítségével)

A Delphi egy nagyszerű, de igen összetett, sok különböző részből felépülő eszköz. E könyv segítségével megismerhetjük a Delphi-t, a hozzá tartozó Object Pascal nyelvet, a komponenseket (megismerkedünk a már létezőkkel, és megtanuljuk, hogyan kell újakat készíteni), az adatbázis-kezeléssel, és a Windows és COM programozás kulcselemeivel.

A fenti témák egyikével sem kell tisztában lennünk ahhoz, hogy megértsük a könyv fejezeteit. Sokat segíthet (különösen az első néhány fejezetnél), ha már ismerjük a Delphi korábbi változatait. A könyv a 4. fejezettől kezdve merül bele a Delphi mélységeibe. Ha még soha nem használtuk ezt az eszközt, akkor kemény munka vár ránk; azonban azok is rengeteg újdonsággal fognak találkozni, akik már dolgoztak az előző verziókkal. A könyv nem kiegészítése a korábbi kiadásoknak: az elejétől a végéig újraírtam, hogy pontosan bemutathassam a környezetet.

A könyv megértéséhez elengedhetetlenül szükséges némi programozási ismeret, legalább egy nyelv és operációs rendszer, ismerete. Azt is fontos tudnunk, hogy mik a változók, és mit jelent a "ciklus" kifejezés. Ha még soha nem írtunk programot életünkben, akkor ez a könyv nem nekünk való. Először biztos programozás-elméleti alapokat kell szereznünk, és csak azután álljunk neki a könyvnek.

A könyv szerkezete

A könyv négy fő részből (kötetenként két-két részből) áll, amelyek a következők:

Az első rész címe „Delphi és Object Pascal", és a Delphi programozás alapjait mutatja be azoknak, akik most ismerkednek a témával, de profiknak is sok érdekes ötlettel szolgál. Az első három fejezet a Delphi fejlesztői környezetével foglalkozik, és csak kevés példát tartalmaz. Az ezután következő fejezetek a Object Pascal nyelvvel foglalkoznak, és tartalmazzák a Pascal nyelv, valamint az objektum-orientált programozás rövid leírását. A rész utolsó fejezete a Vizuális Komponenskönyvtár (Visual Component Library, VCL) használatával foglalkozik.

A második rész címe „Komponensek használata", és a Delphi komponenseivel és formjaival foglalkozik. Itt ismerhetjük meg a standard komponenseket, a Windows 95 vezérlőket, a grafikus eszközöket, az egér- és billentyűzet-események kezelését, menük készítését, a képernyőre rajzolás módszereit, az adatbázis-kezelést, és még sok más dolgot. Ezek a fejezetek mind a Delphi-vel, mind pedig a Windows programozásával kapcsolatos témákat tartalmaznak.

A harmadik rész címe „Komponensek és könyvtárak", és a Delphi komponens- és DLL-fejlesztésével foglalkozik, majd áttérünk a COM és OLE problémáira, a Windows shell kiterjesztésére, az OLE Automation-ra, az ActiveX és ActiveForm fejlesztésre, végül megismerkedünk az Internet programozásának alapjaival.

A negyedik, „Felsőfokú Delphi programozás" című rész olyan témákat taglal, mint a Windows memóriakezelése, erőforrások használata, nyomtatás, fájlkezelés, dinamikus adatcsere (Dynamic Data Exchange, DDE), és a multimédia.

A könyvben a példák központi fontosságúak. Minden téma bevezetése után egy, esetleg több példán keresztül mutatom be annak gyakorlati használatát. Összesen több, mint 300 példaprogram található a könyvben, és a mellékelt CD-n. A legtöbb itt található program igen egyszerű, általában egy témára koncentrálnak, és a bonyolultabb alkalmazásokat is lépésről-lépésre, közbülső állomásokkal építjük fel.

A könyv írása során megpróbáltam teljesen kihagyni azokat a részeket, amelyek megtalálhatók a Delphi Súgójában, így a kötetek nem tartalmaznak referenciaanyagot. Az eljárások és tulajdonságok felsorolása nem csak azért felesleges, mert megtalálható a Delphi dokumentációjában, de azért is, mert a program megváltozásával használhatatlanná válna. Azt ajánlom, hogy a könyvet a Delphi dokumentációjával párhuzamosan olvassuk, és ha számítógép mellett vagyunk, akkor a példákat rögtön ki is próbálhatjuk, ha követjük a könyv útmutatásait.

Mindezek ellenére megpróbáltam mindent elkövetni, hogy a könyv ne csak számítógép mellett legyen érthető és élvezhető; ezt a célt szolgálják a könyv ábrái, és a beillesztett programrészletek, amelyek mindig tartalmazzák a

példaprogramok kulcsfontosságú részeit. A teljes programlistákat csak a mellékelt CD-n találhatjuk meg; választanom kellett a között, hogy több témáról essen-e szó, vagy teljes programlistákat tartalmazzon a könyv, és én a több témát választottam. Ez a kötet így körülbelül ötvennel több példát tartalmaz az előző kiadásnál.

A könyv méreteiből kifolyólag még akkor is ajánlatos átlépnünk bizonyos részeket, ha egyébként a könyveket az elejétől a végéig el szoktuk olvasni. A tartalomjegyzék és a fejezetek bevezetőinek elolvasásával eldönthetjük, milyenlépésekben szeretnénk végighaladni a köteten.

A könyv tartalmaz néhány tördelési konvenciót, amelyek olvashatóbbá teszik a szöveget. A forráskódokat ilyen betűtípussal szedtük (a programlisták formázásáról egyébként a 4. fejezetben esik szó). Vannak különböző típusú megjegyzések is, valamint az Új felirat jelzi, ha valami a Delphi 3 újdonsága.

A mellékelt CD

A CD a könyvben található több mint 300 példaprogramot tartalmazza. Ahogy korábban már említettem, a könyv szövege csupán a példák egy részét tartalmazza, a teljes programok a CD-n találhatók. A példák azonnal ki is próbálhatók, a programok lefordítva, futtatható állományként is rajta vannak a lemezen. Ha módosítani szeretnénk a programokat, másoljuk őket merevlemezünkre, és ott elvégezhetjük a változtatásokat.

Négy különböző módszerrel nézhetjük végig a CD tartalmát:

Másoljuk a példákat merevlemezünkre, távolítsuk el a fájlok "Csak olvasható" jelölését, és nyissuk meg őket Delphi-ben.

Kedvenc böngészőprogramunkkal megnyithatjuk a CD CODEMD3.HTM fájlját, ami a BOOKCODE könyvtárban található, majd követhetjük az itt olvasható utasításokat. Minden forráskód megtalálható HTML formátumban is a lemezen. A példák a Delphi szintaxis jelöléseit használják.

Ugyancsak kedvenc böngészőprogramunkkal nyithatjuk meg a CD CROSSREF.HTM fájlját, ami szintén a BOOKCODE könyvtárban található. Itt az azonosítók – függvények, osztályok, metódusok és tulajdonságok – ábécé sorrendbe szedett listáját találhatjuk meg. Minden azonosítóhoz tartozik egy hivatkozáslista, amely azokat a példaprogramokat tartalmazza, amelyekben az adott elem megtalálható.

A CD CDVIEW című alkalmazásával (amit direkt erre a célra készítettem) is megtekinthetjük a lemez tartalmát. Ezzel a programmal gyorsabban nézhetjük meg és futtathatjuk a példákat, mintha azokat a Delphi-ben nyitnánk meg.

A CD könyvtárstruktúrája igen egyszerű – lényegében minden fejezetnek megvan a maga könyvtára, azon belül pedig minden példának a maga alkönyvtára.

Megjegyzés

Néhány olyan esetben, amikor a példának több változata is van, előfordulhat, hogy a program és az őt tartalmazó alkönyvtár neve nem egyezik meg. Az ilyen esetekben a könyv szövegében mindig a könyvtár nevére, és nem a programéra hivatkozunk. Minden könyvtár csak egy példaprogramot tartalmaz, így nem okozhat különösebb gondot a megtalálásuk.

A könyv jelenlegi kiadásához tartozó példák mellett az előző kiadások programjai is megtalálhatók a CD-n. Sok példa hasonlít egymásra, így különböző megoldásokat láthatunk ugyanarra a problémára, a Delphi verziójától függően.

Megjegyzés

Ezen eszközök nagy része még a Delphi 2-höz készült, ennek oka az, hogy a CD összeállításakor a Delphi 3 még nem volt a piacon. Installálhatjuk őket így is, esetleg keressünk újabb verziókat az Interneten.

Külső cégek eszközei

A mellékelt CD-n külső cégek eszközei is megtalálhatók, többek között különböző komponensek és dokumentációk. Ezek némelyike demo, vagy shareware változat. Természetesen ezeket regisztráltatnunk kell a gyártónál, ha használni kívánjuk programjainkban.

A CD angol nyelvű Delphi magazinokat is tartalmaz, amiket a Windows Súgójával, vagy a mellékelt Acrobat Reader programmal tekinthetünk meg.

Saját eszközeim

A mellékelt CD-n, most első esetben, egy kis gyűjteményt találhatunk az általam készített eszközökből is. Nem kell borzasztó bonyolult programokra

számítani, csupán néhány egyszerű, de hasznos segédeszközre. Ezeknek a programoknak a legfrissebb változatait ingyen letölthetjük saját honlapomról. A lemez Readme fájlja tartalmaz egy rövid leírást ezekről az alkalmazásokról, és a könyv szövegében is említést teszek róluk a megfelelő helyen.

A Readme fájlok

A főkönyvtár és a Tools könyvtár Readme fájljai (ezek a teljes CD és az általam írt eszközök leírását tartalmazzák) mellett a külső cégek eszközeinek leírása is megtalálható az ilyen állományokban. Ezek a fájlok segítségünkre lehetnek, hogy helyesen, hatékonyan, és nem utolsó sorban törvényesen használjuk ezeket a segédeszközöket.

Hogyan érhetjük el az írót?

Ha gondjaink vannak a könyv szövegével vagy példáival, szívesen veszem, ha Ön, kedves olvasó, tájékoztat erről. Közvetlenül elérhető vagyok e-mailen keresztül (a címet lásd lejjebb). A könyv változtatásait és a példaprogramok forrásfájljait megtalálhatjuk a Sybex honlapján:

```
http://www.sybex.com/
```

és a saját honlapomon:

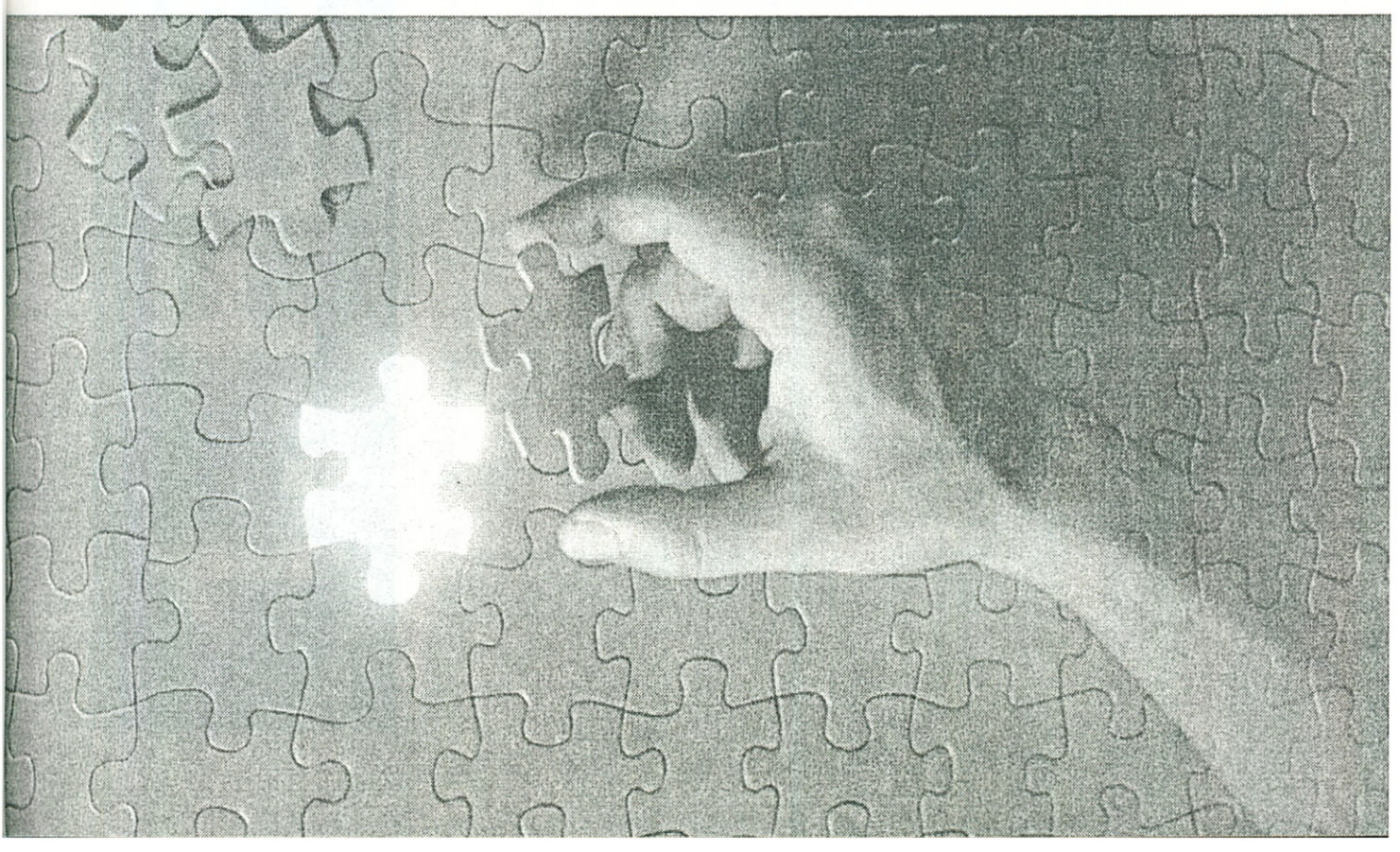
```
http://ourworld.compuserve.com/homepages/marcocantu
```

Saját honlapomon újdonságokat és tippeket is olvashatunk, valamint azt is megtudhatjuk, milyen konferenciákon és szemináriumokon fogok előadásokat tartani (így akár személyesen is találkozhatunk). Az oldal tartalmaz hivatkozásokat más hasonló témájú honlapokra, és itt található személyes Delphi komponens-gyűjteményem is.

Végül pedig mindenképpen szeretném, ha Ön megosztaná velem a véleményét a könyvről: elmondaná, melyik rész tetszett a legjobban, és melyik a legkevésbé. Amennyiben a magyar változathoz kíván véleményt fűzni, kérem, hogy a Kiskapu Kft. címére írjon, a kiskapu@kiskapu.hu postaládába, ha az angol verzióval kapcsolatban van megjegyzése, azt örömmel várom a saját e-mail címemre: marcocantu@compuserve.com

I. Rész

Delphi és Object Pascal



1. FEJEZET

A form egy ablak (és egy ActiveX)

E fejezet tartalma:

- Form készítése
- Komponensek és tulajdonságaik
- Programfordítás
- Események és kód
- A Delphi, mint kétirányú eszköz
- Formból ActiveX

A Form egy ablak (és egy ActiveX)

Ebben a fejezetben elkészítjük első Windows alkalmazásunkat, és úgynevezett ActiveFormunkat Delphivel. Ez a fejezet egy gyors bevezetés a Delphi környezetbe, és bár sok alapvető dolgot tárgyal, tapasztalt programozók is találhatnak benne hasznos tippeket és trükköket, mivel szó lesz néhány új Delphi 3 jellemzőről.

A windowsos programok általában ablakokon alapulnak. Hogyan készítjük el az első ablakunkat? Egy úgynevezett formot használunk majd. Ahogy azt a cím első része már sugallja, a form valójában egy „álruhás" ablak, nincs jelentős különbség a két elnevezés között, legalábbis nem általános szempontból. Később készítünk még egy, az előbbitől némileg eltérő, úgynevezett ActiveFormot. Ez valójában egy formon alapuló ActiveX kontrol, így ezt használhatjuk Web böngészőkben. Az ActiveX kontrol is egy ablak, bár az ilyen különleges ablakoknak különböző képességekkel kell rendelkezniük. A Delphi 3 automatikusan felkínálja ezeket a képességeket a formok egy meghatározott csoportjánál, az ActiveFormoknál.

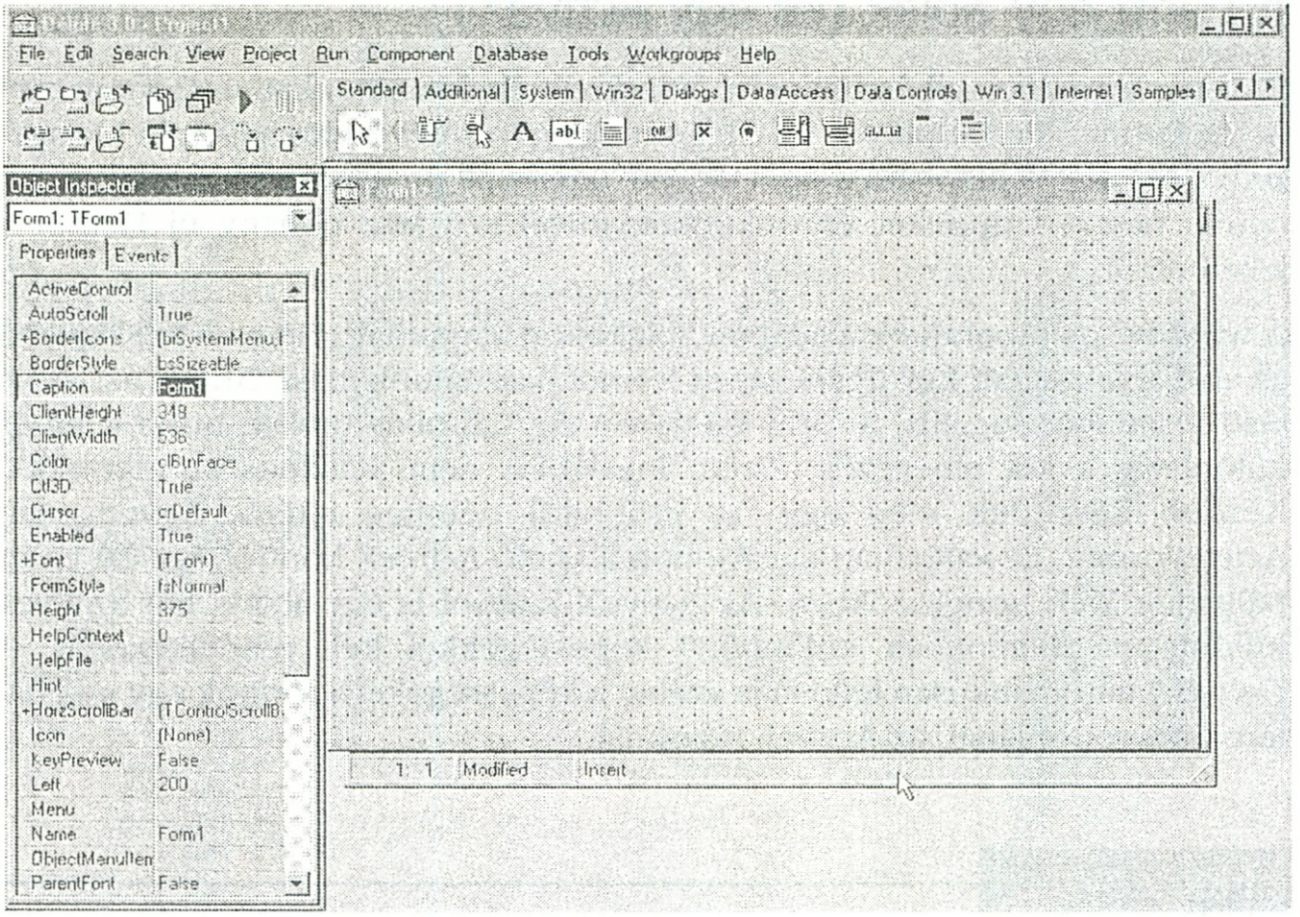
Megjegyzés

Ha közelebbről vizsgáljuk, a form mindig ablak, de ez fordítva nem feltétlen igaz. Néhány Delphi komponens szintén ablak, például a nyomógomb és listadoboz is ablak. Hogy elkerüljük a félreértéseket, a form kifejezés az alkalmazás fő ablakát, vagy a hasonló ablakokat jelent, az ablak kifejezést szélesebb értelemben használjuk.

Első formunk elkészítése

Lehet, hogy már készítettünk formot a Delphivel korábban is, de most újra megmutatjuk, hogyan kell, kiemelve néhány érdekesebb pontot. Egy form elkészítése a rendszer legegyszerűbb művelete: csak el kell indítani a Delphit, és az azonnal készít egy új, üres formot, ahogy az az 1.1. ábrán látszik. Ennyi az egész.

Ha már van nyitott projektünk, akkor a **File** menü **New Application** (új alkalmazás) parancsával zárhatjuk be a régit (lehet, hogy a rendszer rákérdez, hogy elmentse-e a változásokat), és nyithatunk egy új, üres projektet. Íme, bármily hihetetlen, máris kész egy működő alkalmazás. Akár futtathatjuk is, az eszköztárban lévő **Run** (Futtatás) gombbal, vagy a **Run** menü **Run** parancsával, mindkettő eredménye egy szabványos Windows program lesz. Természetesen, ez az alkalmazás nem túl hasznos, hiszen csak egy üres ablakunk van minden funkció nélkül.



1.1. ábra *Ez az üres form a Delphi indításakor keletkezett*

Cím beállítása

Mielőtt elindítjuk az alkalmazásunkat, csináljunk egy apró változtatást. A form címe jelenleg Form1. A felhasználó számára a főablak címe jelenti az alkalmazás nevét is, ezért cseréljük a Form1-et valami kifejezőbbre.

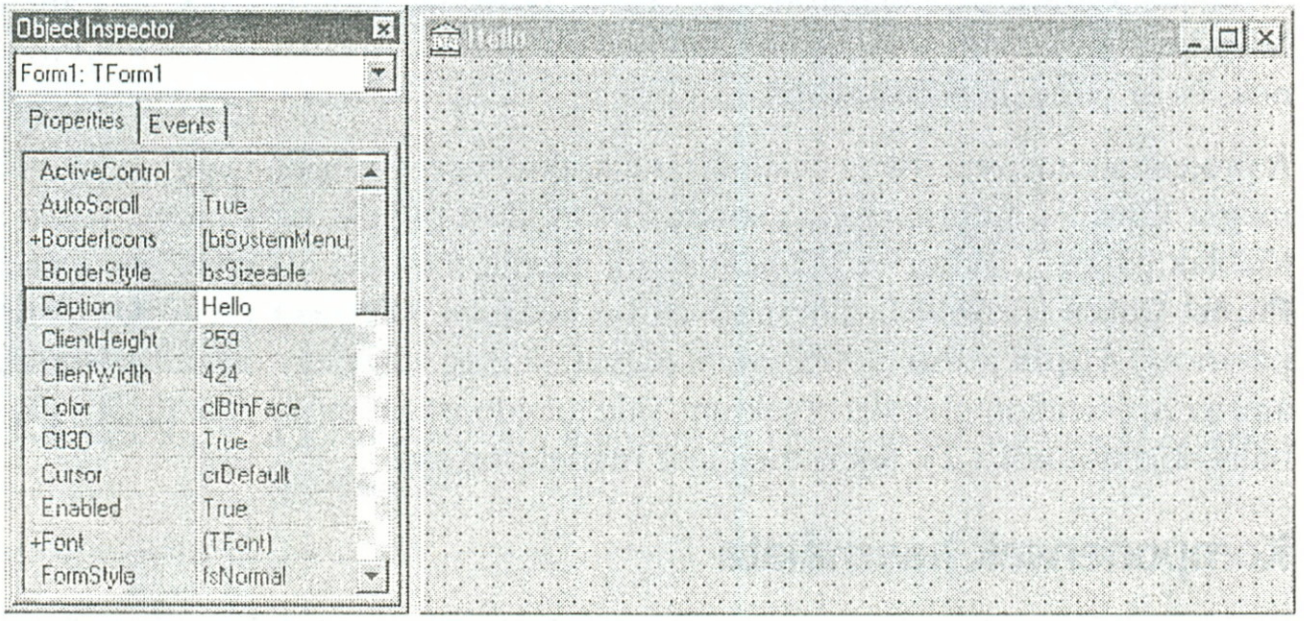
Amikor először elindítjuk a Delphit, az Object Inspector (objektum-figyelő) ablak jelenik meg a form bal oldalán (1.1. ábra). Ha nem lenne ott, akkor a **View** (nézet) menü **Object Inspector** parancsával, vagy az F11 billentyűvel megnyithatjuk. Az Object Inspector a kiválasztott komponens tulajdonságait mutatja. Az ablak egy két oldalból áll. Az első oldal felirata **Properties** (tulajdonságok), másiké **Events** (események), és azoknak az eseményeknek a listáját tartalmazza, amik a formon, vagy a kiválasztott komponensen bekövetkezhetnek.

A tulajdonságok ABC sorrendben szerepelnek, így könnyedén megtalálhatjuk, amit meg szeretnénk változtatni. A címet egyszerűen megváltoztathatjuk a Caption (Címke) tulajdonság átírásával (alapértelmezésben a kiválasztott). Mialatt az új értéket gépeljük, láthatjuk, ahogy az ablak címe is változik. Ha beírjuk azt, hogy „Hello", a form címe azonnal megváltozik, ahogy az az

1.2. ábrán látható. Másik lehetőségként még megváltoztathatjuk az ablak belső megnevezését a Name (Név) tulajdonság átírásával. Amennyiben nem írtunk Captiont, úgy a Name tulajdonság értéke lesz a Caption értéke is.

Megjegyzés

A komponensnek csak néhány tulajdonsága változik, miközben gépelünk, a legtöbb változtatás csak akkor lép életbe, amikor befejeztük a szerkesztést és ENTER-t nyomtunk (vagy más mezőben folytatjuk a bevitelt).



1.2. ábra *Az Object Inspector és a form az új címmel*

Ugyan nem csináltunk túl sok mindent, mégis van egy teljes értékű alkalmazásunk, rendszermenüvel, minimalizáló, maximalizáló és bezárógombokkal. Átméretezhetjük a keret húzásával, mozgathatjuk a címsor húzásával, teljes méretre nagyíthatjuk, vagy minimalizálhatjuk. Működik, de nem túl hasznos. Ha a Tálcára (Taskbar) nézünk, észrevesszük, hogy valami nincs rendjén: ahelyett, hogy a form címe jelenne meg az ikon mellett, a projekt neve szerepel, valami olyasmi, hogy Project1. Ezt úgy javíthatjuk ki, hogy nevet adunk a projektnek, azaz új néven mentjük el.

A form elmentése

Válasszuk ki a **Save Project** (projekt mentése) vagy a **Save Project As** (projekt mentése más néven) parancsot a **File** menüből, és a Delphi megkérdezi a formhoz tartozó forrásállomány és a projekt állomány nevét. Mivel a projekt nevének célszerű kapcsolódnia a form címéhez (Hello), így a form forrását nevezzük HELLOF.PAS-nak, ami a Hello Formot jelképezi, a projekt állományt pedig hívjuk HELLO.DPR-nek.

Sajnos, nem használhatjuk ugyanazt a nevet a projektnek és a formot leíró egységnek; ezeknek minden alkalmazás esetében különböző névvel kell rendelkezniük. A form nevéhez fűzött F bevált gyakorlat volt a 16 bites Delphi verzióban, mivel az állománynevek nem haladhatták meg a 8 karaktert. Delphi 2-től lehetőségünk van hosszú nevek használatára, így például az állomány neve lehetne akár HELLOFORM is. Használhatunk teljesen különböző neveket, vagy hívhatjuk egyszerűen Mainformnak. Ez a gyakorlat azonban oda vezethet, hogy lesz egy csomó projektünk, amikben ugyanolyan nevű ablakok szerepelnek, és ez nagyon zavaró lehet.

A projekt állomány neve az alkalmazás alapértelmezés szerinti címe, ez jelenik meg a Tálcán a program futása során. Emiatt, ha a projekt neve megegyezik a főablak címével, akkor ez a Tálcán is jól jelenik meg. Az alkalmazás címét a Project Option (projekt beállítások) párbeszédablak (**Project** menü **Options** parancsa) **Application** oldalán változtathatjuk meg, vagy egy olyan kódsorral, ami az Application globális objektum Title tulajdonságát módosítja. Erről a globális objektumról a 25. fejezetben lesz részletesen szó.

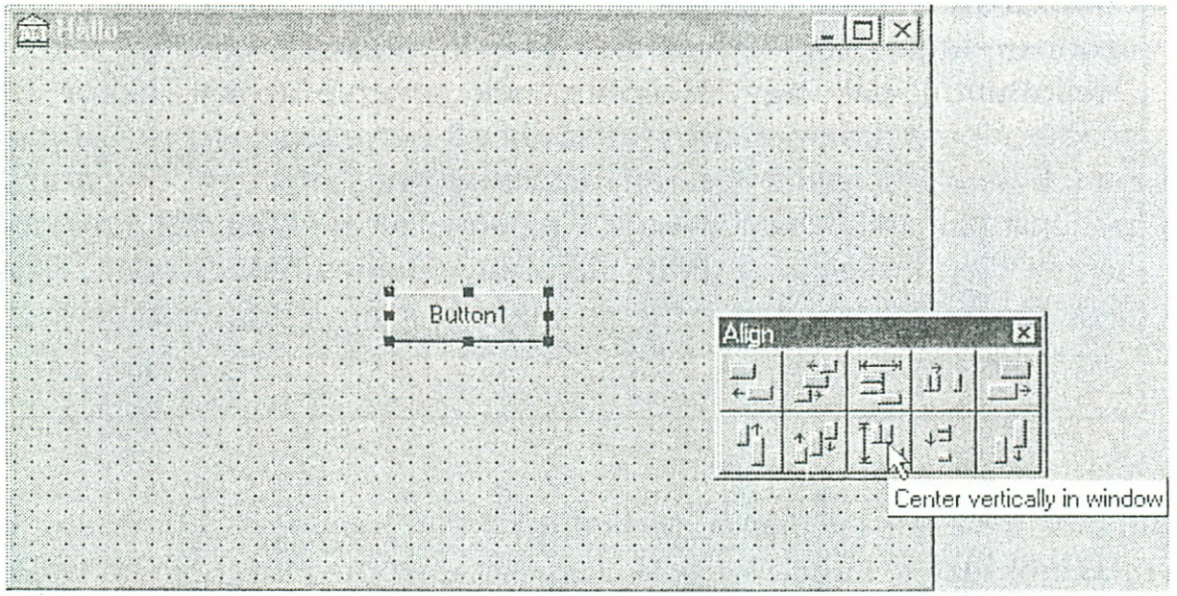
Komponensek használata

Itt az idő, hogy valami hasznosat tegyünk a HELLO formra. A formokat elképzelhetjük, mint komponenstárolókat, mindegyiken több komponens vagy kontrol lehet. A komponenseket a Delphi ablakában, a form felett található Component Palette-ről (komponenspaletta) választhatjuk ki. Négy egyszerű módja van, hogy egy komponenst a formra helyezzünk. Ha például kiválasztjuk a normál nyomógombot a Standard oldalról, akkor a következők bármelyikét tehetjük:

- Kattintsunk a komponensre, vigyük az egeret a formra, nyomjuk le a bal egérgombot, hogy beállítsuk a gomb bal felső sarkát, és húzzuk az egeret a gomb méretének beállításához.
- Válasszuk ki a komponenst, mint fent, és egyszerűen kattintsunk a formra, hogy letegyünk egy alapértelmezés szerinti szélességű és magasságú gombot.
- Kattintsunk kétszer a komponenspalettán, és a kiválasztott komponens megjelenik a form közepén.

- A SHIFT gombot nyomva tartva kattintsunk a komponens ikonjára, hogy több azonos komponenst tehessünk a formra a fenti módszerekkel.

A formunkon csak egy gomb lesz, amit a form közepére teszünk. Ezt megtehetjük kézzel is, a Delphit segítségül hívva. Válasszuk ki a **View** menü **Alignment Palette** (igazítást segítő eszközök) parancsát, és egy eszköztár jelenik meg igazítást segítő ikonokkal (1.3. ábra).



1.3. ábra *A form a középre helyezett gombbal, és az Alignment Palette eszköztárral*

Ez az eszköztár sok műveletet megkönnyít, tartalmaz gombokat kontrolok igazítására, és középre helyezésükre is. A harmadik oszlopban lévő két gombbal a komponenst a form közepére helyezhetjük. Bár a gombot középre helyeztük, amint elindítjuk a programot, átméretezhetjük a formot, és a gomb nem marad középen. Később meglátjuk, hogyan lehet a gombot középen tartani, még átméretezés után is, egy kevés kód segítségével. Most a legelső teendőnk, hogy megváltoztatjuk a gomb címkéjét.

Tulajdonságok megváltoztatása

Akár a form esetében, a gombnak is van Caption tulajdonsága, amit felhasználhatunk a címkéje (a rajta megjelenő szöveg) megváltoztatására. Jobb megoldás azonban, ha a gomb nevét (Name) változtatjuk meg, ami belső tulajdonság és csak a program kódjában szerepel. Ám, amint azt már említettük, ha a gomb nevét megváltoztatjuk, a Caption tulajdonság átírása előtt, akkor a Caption meg fog egyezni a Name tulajdonság értékével. A Name tulajdonság megváltoztatása jó döntés, és célszerű megtenni még a fejlesztés elején, mielőtt túl hosszú programot írunk.

Tipp

Általános gyakorlat elnevezési szabályokat felállítani a különböző komponenstípusokra (ez általában azt jelenti, hogy a komponens nevéhez hozzáfűzzük a komponenstípus teljes nevét, vagy egy rövidebb változatát, mint a „btn" a gombra (button)). Ha minden komponenstípusra különböző előtagokat alkalmazunk (mint a „ButtonHello", vagy a „BtnHello"), akkor az Object Inspector listadobozában a komponensek fajtánként lesznek csoportosítva. Ha azonban utótagokat használunk a komponensek elnevezésekor, „HelloButton"-nak,vagy „HelloBtn"-nek nevezve azokat, akkor az azonos típusú komponensek különböző helyeken lesznek a listában. Ebben a második esetben azonban lehet, hogy egy adott komponenst hamarabb megtalálunk a billentyűzet segítségével, ugyanis ha az Object Inspector az aktív, akkor begépelhetünk egy betűt, és a lista az első olyan elemre ugrik, aminek a neve az adott betűvel kezdődik.

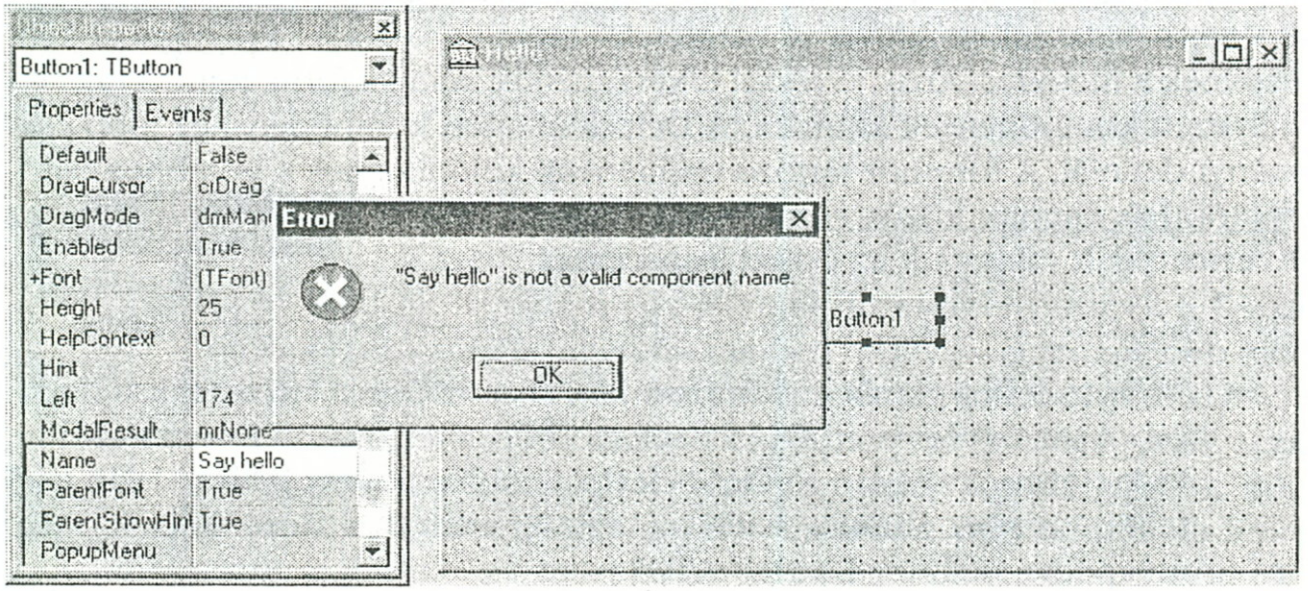
A komponens nevének beállítása mellett gyakran meg kell változtatni a Caption tulajdonságát is. Ennek legalább két oka lehet. Az első, hogy a nevek gyakran követnek valami elnevezési konvenciót (ahogy az a fenti megjegyzésből látható volt), ami nem igazán jó címkének. A második ok, hogy a címke valami leírást ad, amiben gyakran szükség van két vagy több szóra, mint például a „Say hello" gomb. Ha ezt a szöveget Name tulajdonságként próbálnánk használni, akkor a Delphi az 1.4. ábrán látható hibaüzenettel válaszolna.

A Name belső tulajdonság, és a komponensre hivatkozó változó neveként is szerepel, ezért itt a Pascal nyelv változó elnevezési szabályait kell követnünk:

- Az azonosító betűk, számok, és aláhúzásjelek tetszőleges hosszú sorozata (bár csak az első 63 karakter számít).
- Az azonosító első karaktere nem lehet szám, betűnek, vagy aláhúzásjelnek kell lennie.
- Azonosítóban nem lehet szóköz.
- Az azonosítóknál nincs különbség kis és nagybetűk között, de általában az azonosító miden szava nagybetűvel kezdődik, mint a BtnHello-ban. De a btnhello, btnHello, és a BTNHello is ugyanazt az azonosítót jelöli.

Tipp

Az `IsValidIdent` rendszerfunkciót használhatjuk arra, hogy megállapítsuk egy adott karakterláncról, megfelel-e azonosítóként. A CD-n lévő CheckId példa ezt a funkciót hívja meg miközben a beviteli mezőbe gépelünk, jelezve, hogy a karakterlánc jó-e (zöld), vagy nem (piros). Maga a példa kódja elég egyszerű, úgyhogy ezt önállóan is átnézhetjük. Bármilyen kétségünk van, hogy mi lehet komponens név, futtassuk ezt a programot.



1.4. ábra *A hibaüzenet, ami arra figyelmeztet, hogy a Name tulajdonságban nem lehet szóköz*

Itt van azoknak a változtatásoknak az összegzése, amit a gomb és a form tulajdonságain végeztünk. A form struktúrája a példákban úgy jelenik meg, ahogy azután néz ki, miután olvasható formára alakítottuk (ebben a fejezetben később meglátjuk, hogyan kell egy formot szöveggé alakítani). Nem jelenik meg a form teljes leírása (ami gyakran nagyon hosszú), csak a fontosabb elemek, nem szerepelnek továbbá a komponensek pozícióját, méretét leíró sorok, sem a kevésbé érdekes alapértékek. Íme a kód:

```
object Form1: TForm1
  Caption = 'Hello'
  OnClick = FormClick
  object BtnHello: TButton
    Caption = 'Say hello'
    OnClick = BtnHelloClick
  end
end
```

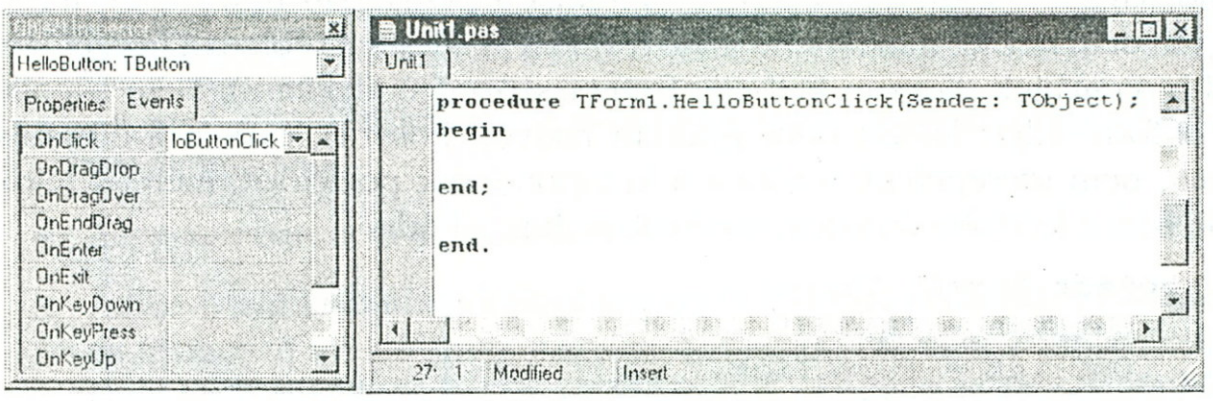
Ez a leírás a komponensek néhány tulajdonságát és eseményét mutatja, az eseményekhez tartozó kódot a következő bekezdésekben láthatjuk. Ha lefuttatjuk a programot, látni fogjuk, hogy a gomb tökéletesen működik. Tehát, ha rákattintunk, a gomb benyomódik, ha elengedjük az egérgombot, a képernyőn lévő gomb is felengedett lesz. Az egyetlen probléma, hogy ha lenyomjuk a gombot, akkor valaminek történni kellene, de most semmi nem változik, mert még nem kapcsoltunk semmilyen tevékenységet az egérkattintáshoz.

Reagálás az eseményekre

Amikor lenyomjuk az egérgombot egy formon vagy egy komponensen, a Windows egy üzenettel informálja az alkalmazást az eseményről. A Delphi az üzenetre válaszul meghívja a megfelelő eseménykezelőt. Programozóként ezeknek a metódusoknak jó részét elkészíthetjük, mind a form, mind a rajta elhelyezett komponensek számára. A Delphi minden komponenshez több eseményt definiál. A formhoz tartozó események különböznek a gombhoz tartozóktól, bár vannak közös eseményeik is, erről meg is győződhetünk, ha az **Events** oldalt választjuk az Object Inspectorból.

Több módszer van arra, hogy a gombhoz OnClick eseménykezelőt készítsünk:

- Válasszuk ki a gombot akár a formon, akár az Object Inspector listadobozából (az úgynevezett Object Selectorból), menjünk az **Events** oldalra, és kattintsunk duplán az OnClick (kattintás) esemény jobb oldalán lévő fehér mezőn, aminek hatására egy új metódus neve jelenik meg, mégpedig az, hogy BtnHelloClick.
- Válasszuk ki a gombot, menjünk az **Events** oldalra, írjunk be egy új nevet az OnClick esemény jobb oldalán lévő üres részre, majd nyomjuk meg az ENTER billentyűt, hogy elfogadtassuk.



1.5. ábra *Az Object Inspector Events oldala, és a szerkesztő ablak, miután elkészítettük a HelloButton komponens OnClick eseményét*

- Ha duplán kattintunk a gombon, akkor a Delphi a komponens alapértelmezett eseményét végzi, ami a gomb esetében az OnClick esemény kitöltése. Egyéb komponensek esetében más alapértelmezett események is lehetnek.

Bármelyiket is választjuk, a Delphi készít egy `BtnHelloClick` (vagy más, általunk beírt) nevű eljárást a form kódjába, és megnyitja a forrásállományt azon a helyen, amint azt az 1.5. ábra mutatja.

A gomb alapértelmezett eseménye a kattintás eseményre válaszoló eljárás elkészítése. Ha nem vagyunk biztosak, hogy mi egy bizonyos komponens alapértelmezett eseménye, nyugodtan kattintsunk rá duplán. Ha olyan eljárásba kerülünk, amire nincs szükség, csak hagyjuk üresen, ugyanis a Delphi által generált üres eljárások azonnal eltűnnek, amint elmentjük az állományt. Más szóval, ha nem írunk bele semmit, akkor megsemmisül.

Tipp

Ha el akarunk távolítani egy általunk írt eseménykezelőt a Delphi forrásból, akkor kitörölhetünk minden hivatkozást, ennél jobb módszer azonban, ha csak a kódot töröljük ki az eljárásból, a deklarációt és a `begin / end` kulcsszavakat meghagyva. Ez megegyezik azzal, amit a Delphi automatikusan generált, amikor úgy döntöttünk, hogy lekezeljük az adott eseményt. Ha elmentjük, vagy lefordítjuk a projektet, a Delphi automatikusan eltávolít minden üres metódust a forráskódból és a form leírásából (beleértve a hivatkozásokat is az Object Inspector **Events** oldalán). Ennek megfelelően, ha meg szeretnénk tartani egy üres eseménykezelőt, írjunk bele megjegyzést, így nem tűnik el.

Most akkor elkezdhetünk néhány utasítást begépelni az eljárást határoló `begin` és `end` közé. Ha még ismeretlen számunkra az Object Pascal nyelv, akkor sem kell aggódni, mivel a legfontosabb jellemzőit megismerhetjük a 4. és az 5. fejezetből. A kód megírása általában olyan egyszerű, hogy nem kell a nyelv szakértőjévé válni ahhoz, hogy elkezdhessünk Delphivel dolgozni. Az alábbi kódból csak a középső sort kell begépelni, de itt az egész eljárás szerepel, hogy látszódjon, hova kell beszúrni:

```
procedure TForm1.BtnHelloClick(Sender: TObject);
begin
  MessageDlg ('Hello, guys', mtInformation,[mbOK], 0);
end;
```

A kód nagyon egyszerű. Egyetlen függvényhívás van benne, a `MessageDlg`, ami megjelenít egy kis üzenetablakot. A függvénynek négy paramétere van, és amint begépeltük a nyitó zárójelet, a Delphi azonnal megjelenít egy segítő ablakot, hogy könnyebben eszünkbe jussanak.

Új!

Ez az új szolgáltatás, az úgynevezett Code Parameters (paraméterek), része a Delphi 3-ban bemutatott Code Insight technológiának. Erről részletesebben a következő fejezetben lesz szó.

Ha a függvény paramétereiről, vagy a jelentésükről több információra van szükségünk, akkor kattintsunk nevére az editorban, és nyomjuk meg az F1-et. Ez behozza a súgó információkat. Mivel ez az első kódunk, itt következik ennek a leírásnak a rövid összefoglalója (a könyv többi részében általában nem ismételjük el a Delphi Súgóban megtalálható információkat, inkább a nyelv és a környezet lehetőségeit bemutató példákra koncentrálunk):

- Az első paraméter az üzenetként megjelenítendő szöveg.
- A második paraméter az üzenetablak típusa. Ezt a következők közül választhatjuk: `mtWarning` (figyelmeztetés), `mtError` (hiba), `mtInformation` (információ), vagy `mtConfirmation` (megerősítés). Mindegyik típushoz a megfelelő címsor tartozik, és egy rá jellemző ikon jelenik bal oldalt meg a szöveg mellett.
- A harmadik paraméter a használni kívánt gombok halmaza. Ezek a következők lehetnek: `mbYes` (igen), `mbNo` (nem), `mbOK` (rendben)), `mbCancel` (mégse), vagy `mbHelp` (súgó). Mivel ez egy halmaz érték, így több mint egy elemet használhatunk. Mindig ügyeljünk a halmaz helyes jelölésére, ami a szögletes zárójelekkel ([és]) történik, még akkor is, ha csak egy értéket használunk, mint a fenti példában (a 4. fejezetben lesz szó a Pascal halmazairól).
- A negyedik paraméter a súgó környezet (az úgynevezett help context), egy szám, ami azt jelzi, hogy a súgó hányadik oldala jelenjen meg, ha a felhasználó F1-et nyom. Ha, mint a fenti példában is, nincs ilyen oldal, akkor egyszerűen írjunk 0-t.

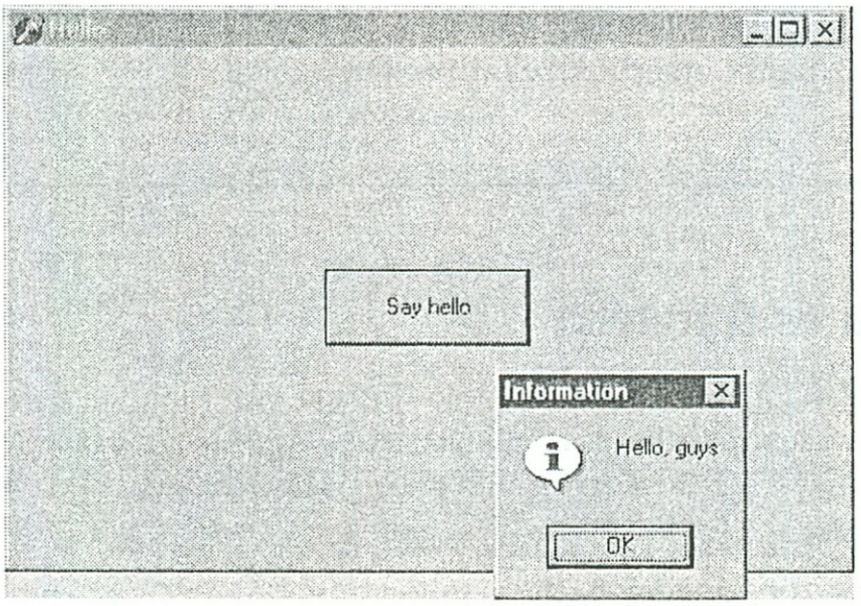
Ennek a függvénynek van visszatérési értéke is, amit figyelmen kívül hagytunk, mivel eljárásként használtuk. Minden esetre érdemes tudni, hogy az

eljárás annak a gombnak az azonosítójával tér vissza, amire a felhasználó kattintott. Ez csak akkor érdekes, ha több, mint egy gomb van az adott üzenetablakon.

Megjegyzés

Azok a programozók, akik még nem ismerik a Pascal nyelvet, különösen azok, akik C-t vagy C++-t használtak, csodálkozhatnak az eljárás és a függvény megkülönböztetésén. Pascalban két külön kulcsszó vonatkozik az eljárás és függvény definícióra, ám a különbség csak annyi, hogy a függvényeknek van visszatérési értéke.

Miután ezt a kódsort beírtuk, a programnak futnia kell. Amikor a gombra kattintunk, az 1.6. ábrán látható üzenetablak jelenik meg.



1.6. ábra *A „Say Hello" gomb megnyomásakor megjelenő üzenetablak*

Akárhányszor a felhasználó rákattint a gombra, az üzenet megjelenik. De mi történik, ha valahova kívülre kattintunk? Semmi. Természetesen néhány plusz kódsorral ezt az eseményt is lekezelhetjük. Csak annyi a teendőnk, hogy a formhoz is írjunk egy OnClick eseménykezelőt. Ehhez válasszuk ki az Object Inspector **Events** lapját, és jelöljük ki a formot. Ezután kattintsunk duplán az OnClick esemény jobb oldalán, és máris a szerkesztő ablak megfelelő helyén találjuk magunkat. Írjuk be a következő `MessageDlg` funkcióhívást:

```
procedure TForm1.FormClick(Sender: TObject);
begin
  MessageDlg ('You have clicked outside of the button',
    mtWarning, [mbOK], 0);
end;
```

Programunk mostani verziójában, ha a felhasználó a gombra kattint, akkor a „Hello" üzenet jelenik meg (1.6. ábra), de ha elhibázza a gombot, egy figyelmeztetést kap.

Feltűnhetett, hogy most két kódsor került be egy helyett. A Pascal fordító figyelmen kívül hagyja az új sorokat, elválasztókat, tabulátorokat, és hasonló nem látható formázó karaktereket; az utasításokat pontosvessző (;) választja el.

Figyelem!

Van egy eset, amikor a Delphi nem hagyja figyelmen kívül a sortöréseket: a karakterláncok nem nyúlhatnak át másik sorba. Nagyon hosszú stringek esetében lehetőség van arra, hogy több különböző, egysoros stringbe vágjuk, majd összekapcsoljuk azokat.

A program fordítása és futtatása

Mielőtt továbbfejlesztenénk a Hello programot, álljunk meg egy pillanatra, és nézzük meg, mi történik, amikor elindítjuk az alkalmazást. Amikor az eszköztár **Run** gombjára kattintunk, vagy a **Run** menü **Run** parancsát adjuk ki, a Delphi a következőket teszi:

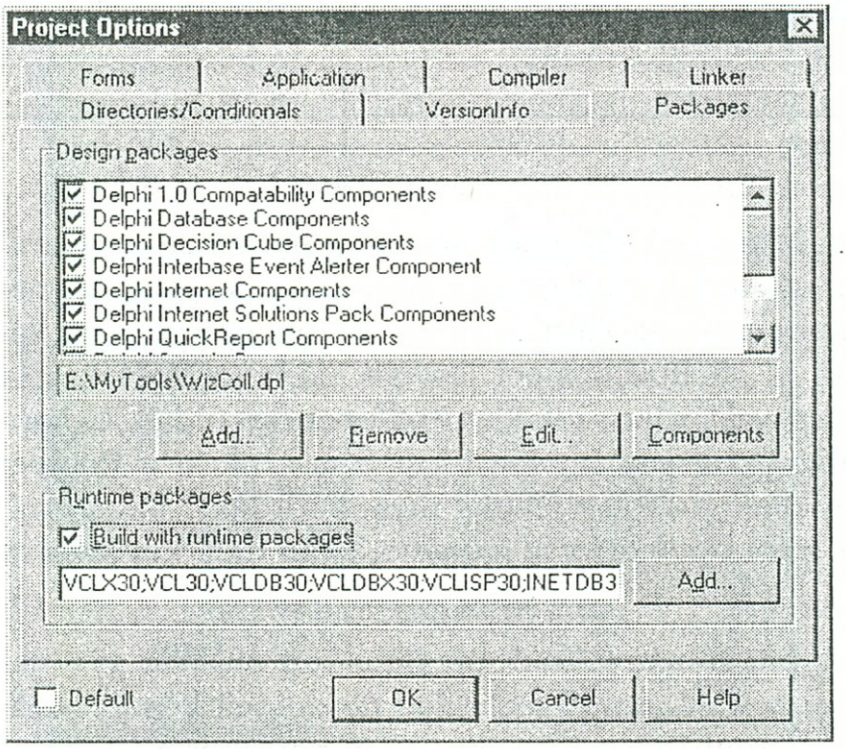
1. Lefordítja a formot leíró Pascal forrásállományt.
2. Lefordítja a projekt állományt.
3. Futtatható (EXE) állományt készít a megfelelő könyvtárak beszerkesztésével.
4. Elindítja a futtatható állományt, általában debug módban.

Új!

A Delphi előző verzióiban a futtatható állomány mindenképpen önálló volt. A Delphi 3 is lehetővé teszi, hogy minden szükséges könyvtárat összeszerkesszünk egy futtatható állományba, de megengedi a különálló, futásidejű csomagok (package) használatát is.

A legfontosabb részlet, hogy amikor a Delphivel lefuttatjuk az alkalmazást, akkor futtatható állomány készül, ezt az állományt elindíthatjuk a Windows Intézőből, vagy a Start menü Futtat... parancsával.

A program szokásos módon történő lefordítása, minden szükséges könyvtár beszerkesztésével, körülbelül 200 kilobájtos futtatható állományt eredményez. Kisebb állományhoz használhatjuk az új Delphi run-time (futásidejű) csomagjait. Ehhez válasszuk ki a **Project** menü **Options** parancsát, menjünk a **Packages** oldalra, és jelöljük be a **Build with runtime packages** (szerkesztés futásidejű csomagokkal) dobozt, ahogy az az 1.7. ábrán látszik. Ennek az opciónak a hatására a futtatható állomány akár 10 kilobájtra is zsugorodhat!



1.7. ábra *A Delphi 3 lehetővé teszi programok használatát, hogy a futtatható állomány mérete csökkenjen*

A csomagok Delphi komponenseket (a Visual Components Library alkotóelemeit) tartalmazó dinamikusan szerkesztett könyvtárak, ahogy azt a 7. és a 21. fejezetben láthatjuk. Csomagok használatával a futtatható állományok sokkal kisebbek lesznek, de nem fognak futni, ha a szükséges dinamikusan szerkesztett könyvtárak (mint amilyen a vcl30.dpl) nincsenek meg a futtató gépen. Az új DPL kiterjesztés a Delphi Package Library rövidítése; ezt a kiterjesztést a Delphi csomagok használják, amik tulajdonképpen DLL állományok, de az új kiterjesztés miatt könnyebb felismerni (és a merevlemezen megtalálni) ezeket.

Ha összeadjuk a dinamikus könyvtárak méretét a kis futtatható állományéval, akkor azt kapjuk, hogy a teljes szükséges lemezterület sokkal nagyobb, mint az amúgy nagyobb, önálló futtatható állomány esetében, ezért a csomagok használata nem mindig javasolt. A Delphi 3 hatalmas előnye a többi fejlesztőeszközzel szemben, hogy könnyen megválaszthatjuk, hogy önálló programot akarunk, vagy kis futtatható állományt futásidejű csomagokkal.

Megjegyzés

A Delphi programok mindkét esetben nagyon gyorsan lefordulnak, és a keletkező alkalmazás sebessége összemérhető a C vagy C++ programokéval. A Delphivel fordított kód sokkal gyorsabban fut (legalább tízszer gyorsabban) mint a hasonló kód interpreteres, vagy pszeudo-kódot készítő eszközöknél.

Sokan nem hiszik el, hogy a Delphi valódi futtatható kódot generál, mivel amikor kis programot futtatunk, a fő ablak szinte azonnal megjelenik, ahogy az néhány interpreteres környezetben megszokott. Hogy erről meggyőződjünk, tegyük a következőt: nyissuk ki az Environment Options párbeszédablakot (a **Tools** menü **Options** parancsával), válasszuk ki a **Preferences** oldalt, és kapcsoljuk be a **Show Compile Progress** opciót. Most válasszuk ki a **Project** menü **Build All** pontját. Egy dialógusablak jelenik meg, a fordítás státuszával. Láthatjuk, hogy ez csak pár másodpercet vesz igénybe, talán még kevesebbet egy gyors gépen.

A Borland Pascal-fordítója a hagyományoknak megfelelően nagyon gyorsan dolgozik a Delphibe ágyazva is. Egy csomó technikai részletnek köszönhetően, ez sokkal gyorsabb, mint bármilyen C++ fordító. Az új Borland C++ Builder fejlesztő környezetben (ami nagyon hasonlít a Delphire) a fordítás több időt vesz igénybe, különösen, amikor először fordítunk le egy alkalmazást.

Delphi fordító nagyobb sebességének egyik oka, hogy a nyelv definíciója egyszerűbb, másrészt a Pascal fordítónak és szerkesztőnek az egységek szerkezetéből adódóan sokkal kevesebb dolga van a könyvtárak vagy más előfordított állományok esetében.

Tulajdonságok futásidejű megváltoztatása

Most pedig térjünk vissza a Hello alkalmazáshoz, készítsük el a program új verzióját, a Hello2-t. Ebben láthatjuk, hogy történik, ha futás közben megváltoztatunk néhány tulajdonságot.

Tipp

Ahhoz, hogy a program új verzióját elmentsük anélkül, hogy az előzőt felülírnánk, zárjuk be a projektet, másoljuk át az összes állományt egy másik mappába, és töröljük le a DSK állományt (ha a Delphi készített ilyet). Ezután nyissuk meg az új verziót és azt módosítsuk. Ha a **File** menü **Save Project As** pontját választjuk, akkor az új projekt a régi forrásállományokra fog hivatkozni, és azokat fogja módosítani is. Ugyanez történhet, ha elfelejtjük letörölni a DSK állományt.

A könyvben (és a mellékelt CD-n is), az összetett példák több verzióját találhatjuk, amint azok lépésről lépésre készülnek, ez érthetőbbé teszi a különböző lépéseket, és megkönnyíti annak kipróbálását, hogy mi nem működik a köztes verziókban. Ennek megfelelően, a mostani példának két változata van: Hello és Hello2. A különböző verziók különböző mappákat kaptak a CD-n.

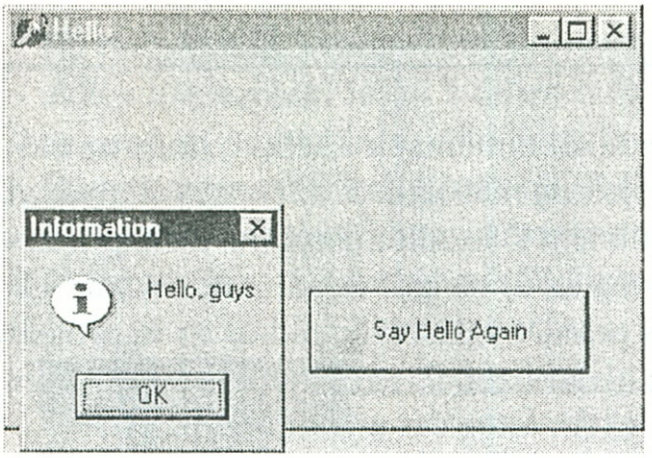
Most megpróbálunk néhány tulajdonságot futásidőben megváltoztatni. Például megváltoztatjuk a HelloButton szövegét „Say hello"-ról „Say hello again"-re, miután a felhasználó először rákattintott. Ezzel együtt szükség lehet a gomb kiszélesítésére, mivel a felirat meghosszabbodott. Ez nagyon egyszerű, csak ki kell cserélni a `HelloButtonClick` eljárásban lévő kódot a következőre:

```
procedure TForm1.HelloButtonClick(Sender: TObject);
begin
  MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
  HelloButton.Caption := 'Say Hello Again';
end;
```

Megjegyzés

A Pascal nyelv a := operátort használja értékadásra, és az = operátort az egyenlőség vizsgálatára. Kezdetben ez zavaró lehet más nyelvet használó programozók számára;. például C-ben vagy C++-ban az értékadó operátor az =, és az egyenlőség vizsgálat az ==. Egy idő után hozzá fogunk szokni, addig pedig, ha sima egyenlőségjelet használunk := helyett, hibaüzenetet kapunk a fordítótól.

Egy tulajdonságot, mint amilyen a Caption is, könnyen meg lehet változtatni futásidőben, egy értékadással. Ennek eredményét az 1.8. ábra mutatja.



1.8. ábra *A gomb új felirata a változás után. Látható, hogy a gomb nincs középen, mert az ablakot átméreteztük – egy probléma, amit hamarosan megoldunk*

A legtöbb tulajdonságot meg lehet változtatni futásidőben is, sőt, néhányat csak akkor. Az utolsó csoportba tartozókat nagyon könnyen fel lehet ismerni onnan, hogy nem jelennek meg az Object Inspectorban, de a komponens leírásánál a Súgóban szerepelnek. Ezek közül a futásidejű tulajdonságok közül néhány csak olvashatóként értelmezett, ami azt jelenti, hogy bár az értékéhez hozzáférhetünk, meg nem változtathatjuk. A tulajdonság definíciókról bővebben a 6. fejezetben lesz szó.

Kód hozzáadása a programhoz

A programunk már majdnem kész, de még van egy megoldatlan probléma, amihez viszont tényleg programozni kell. A gomb kezdetben a form közepén van, de nem marad ott, amikor átméretezzük az ablakot. Ezt a problémát két teljesen különböző módon lehet megoldani.

Az egyik megoldás, hogy a form keretét vékonyra változtatjuk, így azt futásidőben nem lehet átméretezni: állítsuk át a form BorderStyle (keretstílus) tulajdonságát `bsSingle`-re (vékony keret) `bsSizeable`-ről (méretezhető keret). A másik megközelítés, hogy írunk egy programrészt, ami a gombot minden átméretezés után középre mozgatja, tegyük most ezt. Bár eddig úgy tűnhetett, hogy a Delphi programozás csak opciók és vizuális elemek kiválogatásából áll, eljön az idő, amikor kódot kell írni. Ahogy tapasztaltabbakká válunk, egyre több időt fogunk kódírással tölteni.

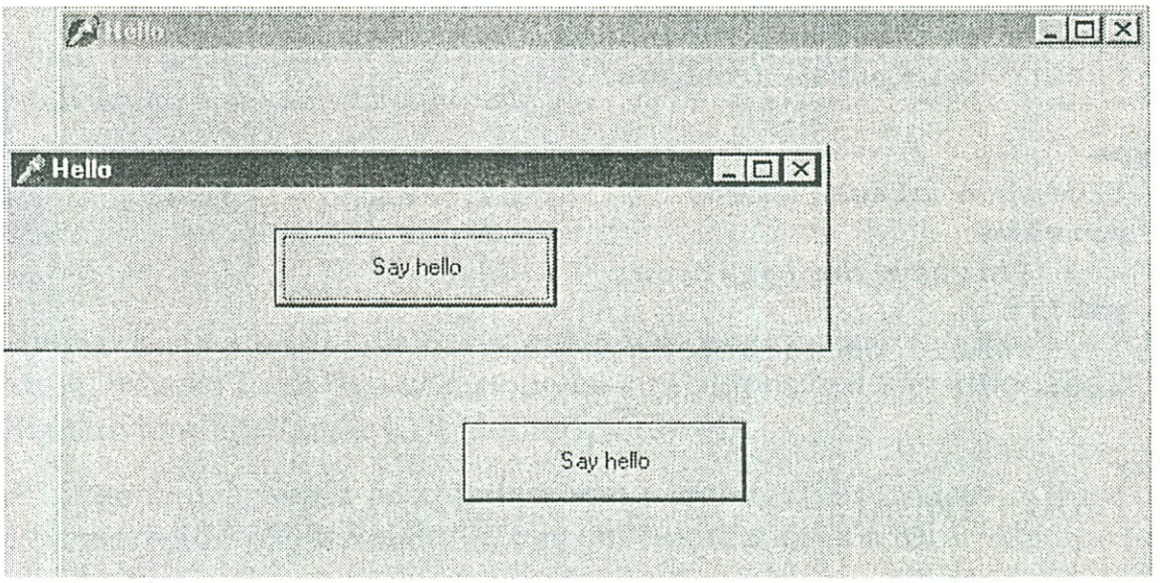
Amikor új kódot adunk a programhoz, az első kérdés, amit fel kell tennünk, az a Hova? Eseményvezérelt környezetben a kód mindig valami esemény következtében fut le. Amikor egy formot átméretezünk, az OnResize esemény következik be.

Válasszuk ki a formot az Object Inspectorból és kattintsunk duplán az OnResize mellett az **Events** oldalon. Egy új eljárás jelent meg a form forrásában, amibe gépeljük be a következő néhány sort:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  BtnHello.Top := Form1.ClientHeight div 2 -
    BtnHello.Height div 2;
  BtnHello.Left := Form1.ClientWidth div 2 -
    BtnHello.Width div 2;
end;
```

A gomb Top (fent) és Left (bal) tulajdonságainak –ez a bal felső sarok pozícióját határozza meg – beállításához a program kiszámítja a keret közepét a belső, vagy kliens terület szélességének, illetve magasságának feleként, majd ebből kivonja a gomb szélességének, illetve magasságának felét. Amennyiben a form Height (magasság) és Width (szélesség) tulajdonságát használtuk volna a ClientWidth (kliensszélesség) és ClientHeight (kliensmagasság) tulajdonságok helyett, akkor az egész ablak közepét vettük volna, beleszámítva a címsort is a felső keretnél.

A példának ez az utolsó verziója már egész jól működik, amint az 1.9. ábrán látható. Ez az ábra a form két verzióját ábrázolja, különböző méretekben. Miután elkészítettünk egy Windows alkalmazást, több példányban elindíthatjuk egy időben az Intézővel, ezzel ellentétben a Delphi környezet csak egy példányt tud futtatni egy programból. Amikor elindítunk egy programot a Delphiben, akkor a beépített hibakeresőt (debuggert) indítjuk, és az nem képes két programot egyszerre futtatni – még ugyanannak a programnak két példányát sem.



1.9. ábra *Programunk végső verziójában a Hello gomb mindig a form közepén marad*

Egy kétirányú eszköz

A Hello példában három kis kódrészletet írtunk, hogy lekezeljünk három különböző eseményt. A kód minden része más-más eljárásba (valójában metódusba, ahogy az majd az 5. fejezetből kiderül) tartozott. De hova kerül a kód, amit írunk? A form forrása egy Pascal forrásállományba kerül, abba, amit HELLOF.PAS-nak neveztünk el. Ez az állomány nő és bonyolódik, nem csak amikor az eseményeket kezelő kódot írjuk, hanem akkor is, ha komponenseket adunk a formhoz. A komponensek tulajdonságai a form tulajdonságaival együtt egy másik állományban kaptak helyet, a HELLOF.DFM-ben.

A Delphi kétirányú eszköz, mert bármit csinálunk a vizuális környezetben, az megjelenik kód formájában is. Semmi nincs elrejtve, semmi nem elérhetetlen. Elérhetjük a teljes kódot, és bár néhol elég összetett, mégis bármibe beleszerkeszthetünk. Természetesen, könnyebb csak a vizuális eszközöket használni, legalábbis ameddig nem válunk tapasztalt Delphi programozóvá.

A kétirányú eszköz azt is jelenti, hogy szabadon változtathatjuk a generált kódot, és utána visszamehetünk a vizuális eszközökhöz, ez azonban csak addig igaz, ameddig betartunk néhány egyszerű szabályt.

A forráskód

Nézzük meg, mit generált a Delphi eddigi tényked éseink eredményeképp. Minden tevékenységnek eredménye van – a Pascal kódban, a form kódjában, vagy mindkettőben. Amikor új, üres projektet kezdünk, az üres formhoz csak kevés forrásszöveg tartozik:

```
unit Unit1;
interface
uses
  SysUtils, Windows, Messages, Classes, Graphics,
  Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}
end.
```

A Unit1 (Egység1) nevű állomány használ néhány másik egységet, definiál egy új adattípust (egy osztályt) és egy új változót (az adott osztály egy objektumát). Az osztály neve TForm1, és a TFormból származik, az objektum neve Form1, és az új TForm1 típusba tartozik.

Megjegyzés

A unitok (egységek) a Pascal program moduljai. Ha új projektet kezdünk, a Delphi készít egy program modult, és egy egységet, ami a fő formot definiálja. Akárhányszor új formot adunk egy Delphi programhoz, új unit keletkezik. Az egységeket a Delphi külön-külön fordítja és szerkeszti a főprogramhoz. Alapértelmezés szerint az egységeknek .PAS, a projekteknek pedig .DPR kiterjesztése van.

Ha átnevezzük az állományt, ahogy arról a példában szó volt, a kód megváltozik, hiszen az egység nevének tükröznie kell az állomány nevét, vagyis ha az állományt HELLOF.PAS-nak neveztük, akkor a kód így kezdődik:

```
unit Hellof;
```

Amint új komponenseket adunk hozzá, a form osztálydeklarációja megváltozik. Például, ha egy gombot adunk a formhoz, az új adattípust leíró kódrészlet a következőre változik:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    ...
```

Miután megváltoztattuk a gomb Name tulajdonságát (az Object Inspectorral) BtnHello-ra, a kód ismét változik:

```
type
  TForm1 = class(TForm)
    BtnHello: TButton;
    ...
```

A néven kívül a többi tulajdonság megváltoztatása nem módosítja a forrásállományt, mivel a form és a komponensek tulajdonságai egy különálló, formleíró állományban találhatók (.DFM kiterjesztéssel).

Az új eseménykezelők hozzáadása van a legnagyobb hatással a kódra; amikor új eseménykezelőt definiálunk, egy sor hozzáadódik a form adattípus definíciójához, egy üres metódustörzs az implementációs részhez, és egy kevés információ bekerül a leíró állományba. A következő lista a Hello példa teljes Pascal forrását tartalmazza:

```
unit Hellof;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
Controls,
  Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    BtnHello: TButton;
    procedure BtnHelloClick(Sender: TObject);
    procedure FormClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.BtnHelloClick(Sender: TObject);
begin
  MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
end;
procedure TForm1.FormClick(Sender: TObject);
begin
  MessageDlg ('You have clicked outside of the button',
    mtWarning, [mbOK], 0);
end;
end.
```

Érdemes megjegyezni, hogy csak egy állomány van a form teljes kódjára, és nem sok kis részlet. Természetesen a kód csak egy része a form leírásának; ez határozza meg, hogy a form és komponensei hogyan reagálnak az eseményekre. A form leírása (a DFM állomány) tartalmazza a form és a komponensek tulajdonságainak értékét. Általánosan úgy mondhatjuk, hogy a forráskód határozza meg a rendszer tevékenységeit, és a form állományok definiálják a rendszer kiinduló állapotát.

A form szöveges leírása

Ahogy azt már említettük, a forráskódot tartalmazó PAS állomány mellett van egy másik állomány is, ami a formot, annak tulajdonságait, a komponenseket, és a komponensek tulajdonságait írja le. Ez a bináris DFM állomány, amit nem

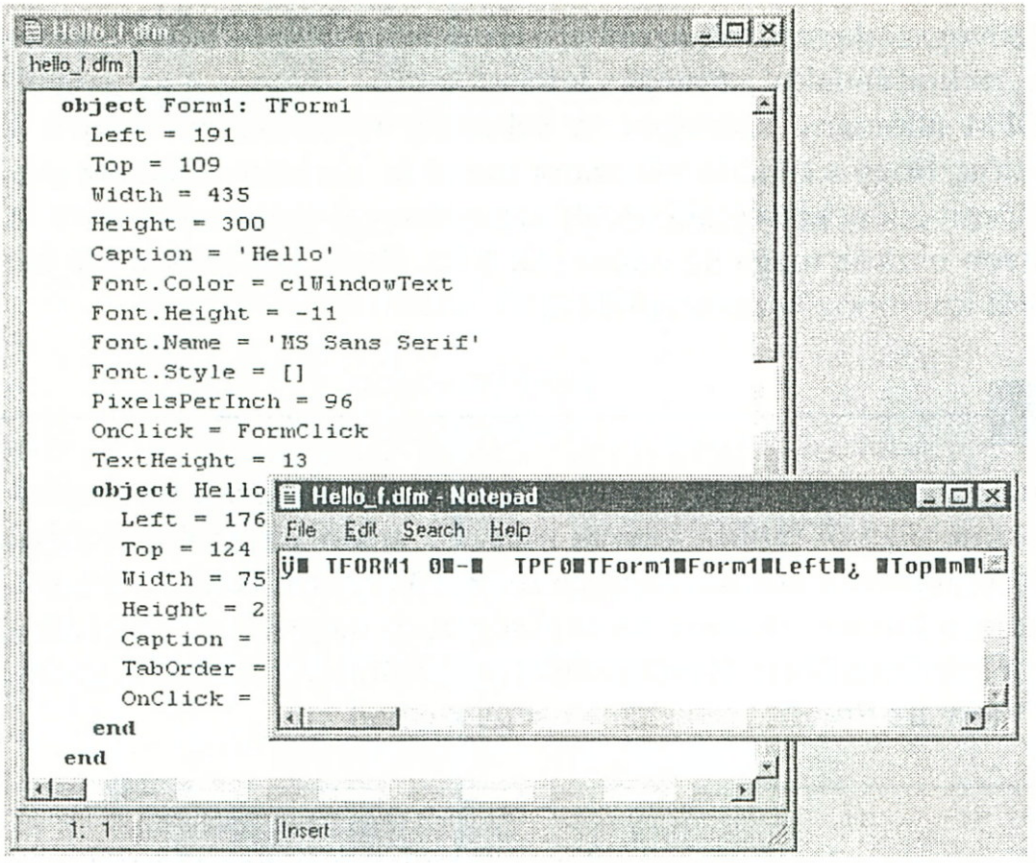
lehet elolvasni szövegszerkesztővel, azonban, ha a fájlt a Delphi kódszerkesztőjébe töltjük be, akkor átalakul szöveges leírássá, ami – tévesen – azt sugallhatja, hogy a DFM állomány szöveges. A sebesség növelése érdekében a Borland úgy döntött, hogy a bináris változatot menti le, de biztosította az eszközöket, hogy könnyedén konvertálhassuk azt szöveggé vagy szövegből. A legegyszerűbb ilyen eszköz maga az editor. Az 1.10. ábrán láthatjuk a fájl két megjelenését a két különböző szerkesztőben.

Tipp

A form szöveges leírását egyszerűen megnyithatjuk, ha a formtervező gyorsmenüjéből (akkor jelenik meg, ha a formon jobb gombbal kattintunk tervezés közben) kiválasztjuk a **View as Text** parancsot. Ez bezárja a formot, elmenti, ha szükséges, és megnyitja a DFM állományt a szerkesztőben. Később visszatérhetünk a formhoz a gyorsmenü **View as Form** parancsának segítségével.

Hogy megértsük, mi van a DFM állományban, nézzük a következő listát, ami a Hello példa első formjának szöveges leírását mutatja. Ez pontosan az a lista, amit akkor láthatunk, ha a form saját menüjében a **View as Text** parancsot választjuk:

```
object Form1: TForm1
  Left = 201
  Top = 110
  Width = 435
  Height = 300
  ActiveControl = BtnHello
  Caption = 'Hello'
  Font.Charset = ANSI_CHARSET
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OnClick = FormClick
  PixelsPerInch = 96
  TextHeight = 13
  object BtnHello: TButton
    Left = 161
    Top = 116
    Width = 105
    Height = 41
    Caption = 'Say hello'
    TabOrder = 0
    OnClick = BtnHelloClick
  end
end
```



1.10. ábra *A DFM állomány a form bináris leírását tartalmazza, ami a Delphi editorával szöveggé alakítható*

Ezt a kódot összehasonlíthatjuk az előzővel, amin a form és a komponensek főbb tulajdonságait és jellemzőit vizsgáltuk. Amint az a listából látszik, a form szöveges leírása több objektumot tartalmaz (ebben az esetben kettőt), különböző szinteken. A Form1 objektum tartalmazza a BtnHello objektumot, amint a lista tagolásából is azonnal látszik. Mindegyik objektumnak vannak tulajdonságai, és eseményekhez kötött metódusai (ebben az esetben az OnClick).

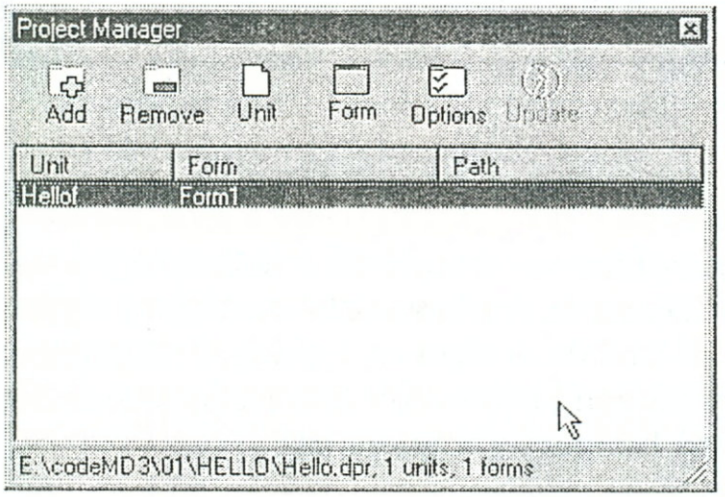
Miután megnyitottuk ezt az állományt a Delphiben, szerkeszthetjük a form szöveges leírását, de ez nagyon nagy odafigyelést igényel, mivel amint elmentjük az állományt, az azonnal visszaalakul bináris formátumra, így ha hibáztunk változtatás közben, a fordítás hibaüzenettel leáll, és ki kell javítani a DFM állomány tartalmát, mielőtt újra megnyithatnánk a formot a szerkesztőben. Emiatt, ameddig nem szerzünk kellő gyakorlatot a Delphi programozásban, kerüljük a szöveges változat közvetlen szerkesztését.

Tapasztalt programozók több ok miatt használják a form szöveges leírását: egyrészt nagy projekteknél a form szöveges leírása hasznos dokumentációs eszköz, jól használható mentési célokra (ha valaki elrontotta a formunkat, a két szöveges leírás összehasonlításából rájöhetünk, hogy mi változott), és verziókövető eszközökkel is használható. Ezek miatt a Delphinek van egy

parancssoros eszköze, a CONVERT.EXE, ami képes formokat szöveggé alakítani, és fordítva. Amint azt a következő fejezetből láthatni fogjuk, ugyanez a folyamat megy végbe, amikor a Vágólapra másolunk egy komponenst a formról.

A projekt-állomány

A formot leíró két állomány (PAS és DFM) mellett van még egy harmadik állomány is, ami az alkalmazás felépítéséért felelős, ez a Delphi projekt állomány (DPR). Ez az állomány automatikusan jön létre, és nagyon ritkán kell megváltoztatnunk, különösen kis programoknál. Ha meg kell változtatnunk a projekt viselkedését, arra alapvetően két módszer van: használhatjuk a Delphi Project Managert (1.11. ábra) és beállíthatunk néhány opciót, vagy manuálisan szerkeszthetjük a projekt forrását.



1.11. ábra *A Delphi Project Manager lehetővé teszi, hogy egységeket adjunk a projekthez, ezzel megváltoztatva a projekt forrásállományát*

A projekt állomány valójában egy Pascal forrásfájl, ami a program általános szerkezetét és az induláshoz szükséges kódot írja le.

```
program Hello;

uses
  Forms,
  Hellof in 'HELLOF.PAS' {Form1};
{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Ezt az állományt a **View** menü **Project Source** (projekt forrás) parancsával nézhetjük meg. Másik megoldásként használhatjuk a **Select unit** (egység kiválasztása) gombot a SpeedBar-on (a Delphi felső eszköztárán), vagy az ennek megfelelő menü parancsot, a **View** menü **Units** pontját. Ennek hatására a Delphi megjelenít egy párbeszédablakot, amiben a projekt forrásállományai szerepelnek, innen kiválaszthatjuk a projekt állományt (amit a példában Hellonak neveztünk), vagy bármelyik másikat, amit meg akarunk nézni.

Megjegyzés

Bővebben olvashatunk erről a kódról a 25. fejezetben, ahol a globális Application objektumról lesz szó. A 2. fejezetben érintőlegesen arról is lesz szó, hogy hogyan kezelhetjük a projektet forrásállományának manuális megváltoztatása nélkül.

ActiveForm készítése

Most, hogy elkészítettünk egy egyszerű formot, átléphetünk a Delphi 3 programozás egy új területére, az ActiveX és az ActiveFormok készítéséhez. Ugyan a 21. fejezetig nem megyünk bele az OLE és az ActiveX technikai részleteibe, most megnézzük, hogy milyen egyszerűen alakíthatjuk át a Hello formunkat ActiveFormmá, és nézhetjük meg egy böngészőprogrammal.

Figyelem!

ActiveX és ActiveForm támogatás csak a Delphi Professional és Client/Server változatába került be.

Az ActiveX egy olyan kontrol, ami megvalósít néhány COM interfészt (amint azt a könyv 3. részében láthatjuk), az ActiveForm pedig egyszerűen egy ActiveX, amit Delphi formból készítettünk. ActiveX komponenseket már sok nyelvben és sok különböző fejlesztőeszközben lehet készíteni.

Az egyik ok, amiért az ActiveX technológia olyan népszerű az, hogy az így elkészített kontrolokat és formokat beágyazhatjuk egy HTML állományba, és Internet-böngészőben is megnézhetjük. Ha a honlapunkon ActiveX kontrolokat vagy formokat helyezünk el, akkor az oldalunkra ellátogatók automatikusan letöltik az alkalmazást, és saját gépeiken futtatják.

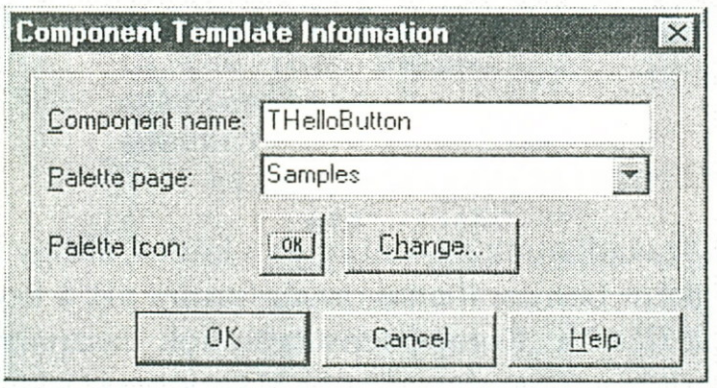
A 23. fejezetben részletesen megnézzük ezt az új technológiát, és meglátjuk, hogyan használhatjuk a Delphit ActiveX kontrolok készítésére, a 24. fejezetben pedig a Delphi Internet-programozási lehetőségeit fogjuk felfedezni. Most azonban csak gyorsan végigmegyünk az első ActiveFormunk elkészítéséhez szükséges lépéseken.

Komponens-sablonok használata

Készítenünk kell egy új alkalmazást, ami ugyanolyan, mint a Hello program: ugyanolyan gomb van benne, és ugyanolyan eseménykezelők. Ahogy azt látni fogjuk a következő fejezetben, lehetőség van egy komponenst a Vágólapra másolni, és onnan egy másik formra tenni, így tökéletes másolatot készítve, de ha így teszünk, akkor csak a komponens tulajdonságait másoltuk át, az eseményeit nem.

A Delphi 3 lehetővé teszi, hogy egy vagy több komponenst lemásoljunk, és komponens sablonokként felhasználjunk, ezzel a módszerrel a komponens eseményeihez kapcsolódó kódot is átvihetjük.

Nyissuk meg a Hello példát, vagy bármelyik másikat, válasszuk ki a sablonként használni kívánt komponenst (vagy komponenseket) és azután válasszuk ki a **Component** menü **Create Component Template** (komponens sablon készítése) parancsát. Ez megnyitja a lentebb látható Component Template Information (komponens sablon információk) párbeszédablakot. Itt beírhatjuk a sablonnak és annak a Component Palette oldalnak a nevét, amelyre komponensünk ikonját fel szeretnénk tenni.

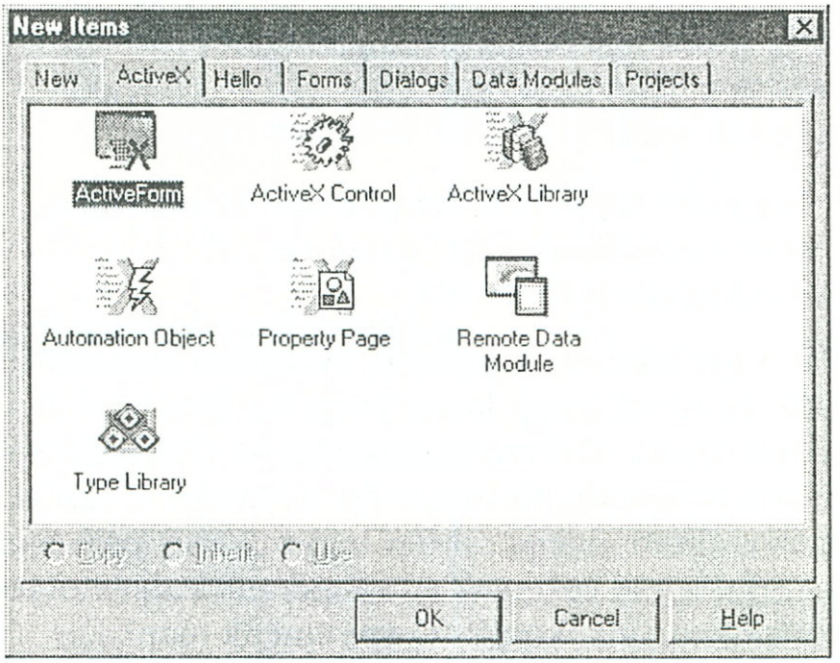


ActiveX könyvtár és ActiveForm

Az ActiveX formot két lépésben készíthetjük el. Először nyitunk egy új projektet, egy ActiveX könyvtárat. Egyszerűen zárjuk be a mostani projektet, válasszuk a **File** menü **New** parancsát, menjünk az **ActiveX** oldalra, és válasszuk az Active Library-t. A Delphi új projektet készít, ami a library (könyvtár) kulcsszóval kezdődik. Amint azt a 20. fejezetben, a DLL programozás tárgyalásakor

látni fogjuk, ez azt jelenti, hogy a fordítás végeredménye nem program, hanem könyvtár lesz. Most már elmenthetjük a projektet egy tetszés szerinti mappába (a CD-n ez az Xform mappa, és a projektnek is ugyanez a neve).

Most hozzáadhatunk egy ActiveFormot a könyvtárunkhoz: válasszuk ki a **File** menü **New** parancsát, menjünk az **ActiveX** oldalra (1.12. ábra), de most ActiveFormot válasszunk. A megjelenő ActiveX Control Wizard párbeszédablakban írjuk be az új ActiveX nevét (a példában Xform1), az állomány nevét (a példában XformF.Pas), és hagyjuk a jelölőnégyzeteket eredeti állapotukban. Kattintsunk az **OK**-ra, és az új form megjelenik a projektben.



1.12. ábra *A File menü New parancsával megnyitott Object Repository ActiveX oldala*

Most, hogy megvan a formunk, hozzáadhatjuk a gombot és a kódot, ahogy azt korábban tettük. Másik lehetőségként bemásolhatjuk az új formra azt a komponens sablont, amit készítettünk (lásd a „Komponens sablonok használata" megjegyzést): egyszerűen válasszuk ki a **Samples** oldalt a komponenspalettán (vagy azt az oldalt, amit használtunk), és válasszuk ki a sablonunkat. Ez átmásolja a komponenst a formra, az OnClick eseményhez tartozó kóddal együtt. Most megpróbálhatjuk lefordítani a programot a **Project** menü **Compile** parancsával. Ne próbáljuk azonban futtatni, mert a könyvtárak nem futtathatók. A programfordítás egy OCX kiterjesztésű állományt eredményez.

A **Project** menü **Web Deploy Options** parancsánál megadhatjuk a szükséges információkat egy HTML példaállomány elkészítéséhez. A tényleges HTML fájl

a **Project** menü **Web Deploy** parancsának hatására keletkezik. A Web Deploy Options párbeszédablak számára szükséges információk a következők:

- *TargetDir:* az a mappa, ahova az OCX állomány (a lefordított ActiveForm projekt) kerül. Ez lehet a Web szerver egyik mappája, vagy egy másik mappa saját lemezünkön, helyi használatra.
- *Target URL:* az OCX állomány URL-je. Ez lehet egyszerűen a komponenst használó HTML-hez viszonyított elérési út. Ha a HTML és az OCX állományok azonos könyvtárban vannak, akkor hagyjuk üresen ezt, vagy gépeljünk be egy pontot útvonalnak (ami az aktuális könyvtárat jelenti).
- *HTML Dir:* az a mappa, ahova a Delphi a példa HTML oldalt tenni fogja.

A többi oldalt is kitölthetjük a szükséges csomagfájlokkal, a tömörítési és biztonsági opciókkal, és így tovább. A Web Deploy szolgáltatás azonban csak a Delphi 3 Client/Server változatában szerepel, így ha Professional változatunk van, kézzel kell elkészíteni a HTML állományt.

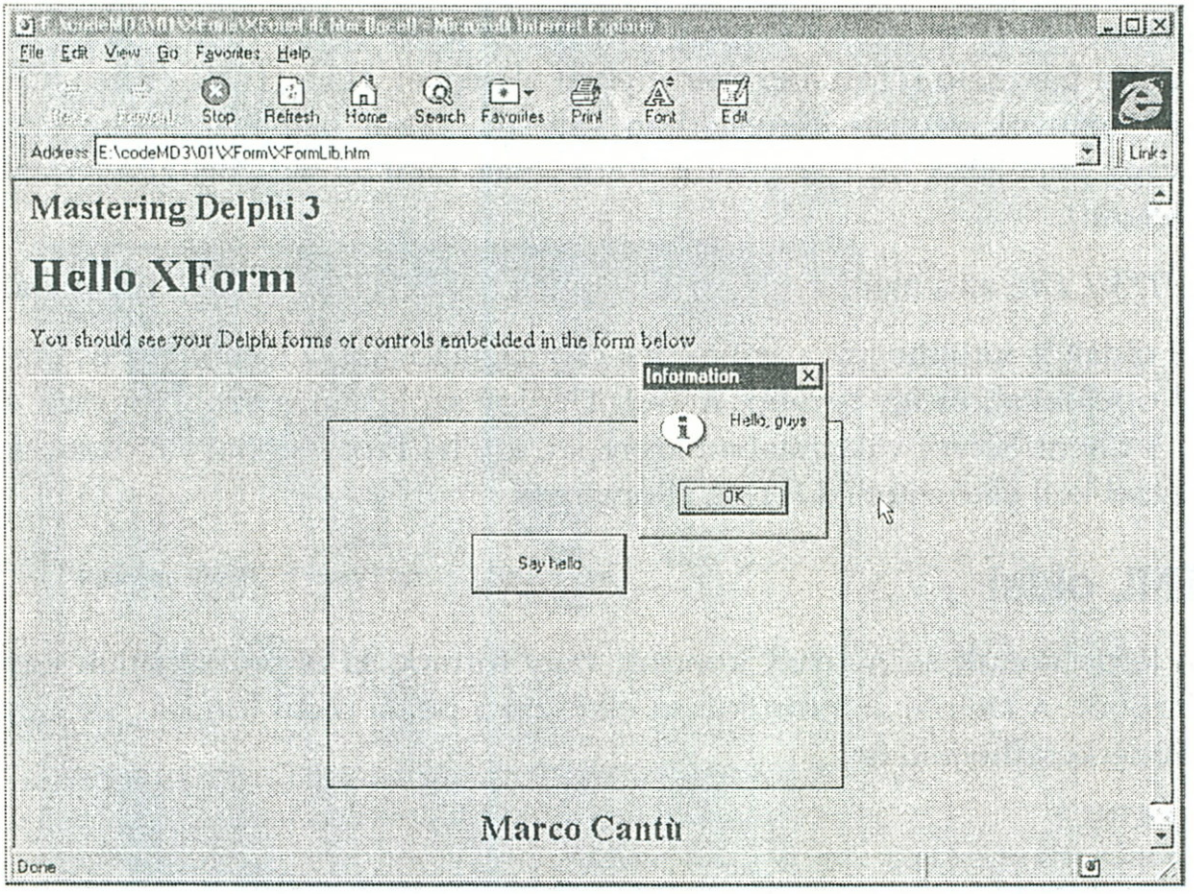
A HTML oldal

Miután lefordítottuk az ActiveX kontrolt vagy formot, ki is próbálhatjuk egy böngészőből. A Delphi automatikusan elkészíti a példa oldal forrását, de egy kicsit átszerkeszthetjük, így:

```
<HTML>
<H2> Mastering Delphi 3 </H2>
<H1> Hello XForm </H1>
You should see your Delphi forms or controls embedded
in the form below.
<HR><center>
<OBJECT
    classid="clsid:367EDB03-8685-11D0-98D0-
        444553540000"
    codebase="file:XForm.ocx#version=1,0,0,0"
    width=350
    height=250
    align=center
    hspace=0
    vspace=0
>
</OBJECT>
<H2> Marco Cantù </H2>
</HTML>
```

Most ezt a HTML oldalt (vagy az Xform.htm állományt a CD-ről) elég csak bemásolni a lefordított OCX állományt tartalmazó mappába. Természetesen kerülhetnek különböző mappákba is, de ebben az esetben a HTML fájlban

módosítani kell az OCX állomány URL-ét, a `codebase="file:..."` parancsban. Mikor ez megvan, a Microsoft Internet Explorerrel (vagy bármilyen más ActiveX-et támogató böngészővel) megnézhetjük az eredményt. A példa az 1.13. ábrán látható.



1.13. ábra *Az ActiveFormunk, ahogy az a Microsoft Internet Explorerben látható, miután megnyomtuk a gombot*

Figyelem!

A Microsoft Internet Explorer az alapbeállítások mellett nem mutatja meg az ActiveX komponenseket, ha nincsenek megfelelő biztonsági kóddal ellátva. Ezt részlegesen ki tudjuk kapcsolni, ha a Safety Levelt (biztonsági szintet)a Security options-nél (biztonsági opcióknál) Medium-ra (közepesre) állítjuk, ilyenkor ugyanis rákérdez, hogy megmutassa-e a nem ellenőrzött forrásokról érkező ActiveX kontrolokat.

Hogyan tovább?

Ebben a fejezetben elkészítettünk egy egyszerű programot, gombot tettünk rá, és néhány alapvető eseményt is lekezeltünk, olyanokat, mint a kattintás vagy az átméretezés. Ezután az egyszerű formunkból ActiveFormot készítettünk, és HTML oldalhoz kapcsoltuk. Megnéztük még, hogy hogyan lehet elnevezni az állományokat, projekteket, és komponenseket, és hogy ez hogyan befolyásolja a program forrását. Megnéztük az elkészült programunk forrását, bár, mivel, lehet, hogy még nem ismerjük eléggé az Object Pascalt, így előfordulhat, hogy nem értettünk minden részletet.

Mielőtt bonyolultabb példákhoz kezdünk, fel kell fedeznünk a Delphi fejlesztőkörnyezetet, és mélyebben kell tanulmányoznunk az Object Pascal nyelvet, ezek a könyv első része többi fejezetének témái. A könyv második részében a VCL komponenseket nézzük meg, míg a harmadik részben visszatér az OLE, ActiveX, és az Internetes fejlesztés.

E fejezet példái bemutatták, hogy a Delphit nagyon könnyű használni, most pedig elkezdjük felfedezni a színfalak mögött meghúzódó bonyolult mechanizmusokat. Meglátjuk, hogy a Delphi nagyon összetett eszköz, bár arra is használható, hogy programokat írjunk gyorsan és egyszerűen.

2. FEJEZET

A Delphi-környezet és jellemzői

E fejezet tartalma:

- A Delphi menüi és eszköztárai
- A formtervezés eszközei
- A kódszerkesztő
- Code Insight
- Projekt menedzsment és fordítás
- Eszközök lefordított programokhoz
- Delphi forrásfájlok

A Delphi-környezet jellemzői

Egy vizuális programozási eszköznél, mint például a Delphi, fontos szerep jut a fejlesztői környezetnek, ez időnként fontosabb, mint a programozási nyelv, amit a fordító vagy az interpreter használ. Ezért érdemes időt szakítani ennek a fejezetnek, vagy legalábbis a Delphi 3 által nyújtott új szolgáltatásokat részletező részek elolvasására.

E fejezetben nem fogjuk a Delphi összes szolgáltatását és menüpontját megbeszélni, ehelyett egy átfogó képet adunk, ami segítségünkre lesz a környezet néhány olyan jellemző vonásának feltárásában, amelyek esetleg nem magától értetődőek, miközben tippekkel is segítjük a Delphi használatának elsajátítását. Később további információkat is találhatunk különböző parancsokról és műveletekről.

A Delphi különböző verziói

Mielőtt mélyebbre ásnánk a Delphi programozói környezetének részleteibe, tegyünk egy kis kitérőt két kulcsfontosságú gondolat erejéig:

- Az alap verzió (Standard kiadás) kezdőknek, és a Delphit ritkábban használó programozók részére készült. Megtalálható benne az összes szolgáltatás, ami a programíráshoz szükséges a Delphi for Windowsban, és magába foglal néhány újat is, ami a korábbi verziókban még nem volt megtalálható. Az új Delphi 3 Standard szolgáltatásai közé tartoznak új komponensek, a Code Insight, a QuickReport új verziója, COM támogatás, az új csomag-architektúra, komponens sablonok, DLL Debugging, és a MS Access támogatás a Borland Database Engine-be építve.
- A második szint (Professional kiadás) hivatásos fejlesztőknek készült. Az összes fent említett szolgáltatás megtalálható benne, plusz a TChart komponens, a VCL forráskódja, ActiveX és ActiveForm fejlesztés támogatása, Internet Solution Pack, ODBC kapcsolat, Database Explorer, Install Shield Express, és az Open Tools API.
- A Delphi teljes (Client/Server Suite kiadás) verziója a kliens/szerver alkalmazások fejlesztőinek készült. Ez tartalmazza a Professional verzió szolgáltatásait, valamint a következő eszközöket és komponenseket: Decision Cube CrossTab komponenst, kiterjedt Web szerver támogatást, SQL Links for native Client/Server BDE Connection-t, BDE Driver Development Kit-et, Open Environment's OLEnterprise-t és más eszközöket az applikáció felosztásra, SQL Monitort, Visual Query Buildert, CASE Tool Expertet és a Data Pump Expertet.

Ez a könyv magába foglalja a három verzió szolgáltatásait, ámbár készítésekor a fő irányadó a Professional verzió volt. Ha nekünk a Standard verzió van meg,

és olyan szolgáltatás leírását látjuk, amely nem működik, akkor valószínűleg a problémát a Delphi 3 verziói közötti különbség okozza. Mivel a mellékelt CD tartalmazza az összes példa futtatható fájljait, a legtöbb ilyen szolgáltatás működését megérthetjük ezek segítségével.

Foglalkozunk a Client/Server Suite néhány szolgáltatásával is, főleg a 18. fejezetben, habár ebbe a könyvbe nem fért bele az összes ide vonatkozó új funkció leírása. Azonkívül a verziók különböző képességei miatt a menük struktúrájában is vannak különbségek. Például a Client/Server Suite tartalmaz egy verziókövető rendszert (version control system), az Intersolv PVCS Version Managementjét, amely egy új legördülő menüt, a Workgroupsot adja a menüsorhoz.

A különböző kiadásokban a Delphi környezetének beállítására több mód is van. Megváltoztathatjuk a SpeedBar gombjait, új parancsokat adhatunk a **Tools** menühöz, elrejthetünk ablakokat vagy elemeket, vagy átméretezhetjük és mozgathatjuk őket. A könyvben található illusztrációknál megpróbáltam csak az alapértelmezett felhasználói felületet használni (ahogy az megjelenik), habár megvannak a saját beállításaim is, és általában felinstallálok sok „add-on"-t (kiegészítő funkciót) is, melyek feltűnhetnek néhány képen.

Segítség kérése

Most már igazán elkezdhetjük a barangolást. Az első környezeti elem, melyet megvizsgálunk, a Help (Súgó) rendszer. Mivel ez a könyv a Delphi programozást példák során keresztül tanítja a komponensek tulajdonságainak és metódusainak listázása helyett, az ilyen típusú kiegészítő információkat a Súgóban találjuk meg.

Tipp

Amikor F1-et nyomunk, akkor a Delphi nem egy pontos egyezést keres a Súgó keresési listájában. Helyette megpróbálja megérteni, hogy mit kérdezünk. Például, ha a kurzor a forráskódban a Button1 komponens nevén áll, amikor F1-et nyomunk, akkor a Delphi Help rendszere automatikusan a TButton osztály leírását nyitja meg, mivel valószínűleg ez az, amit keresünk. Ez akkor is működik, ha a komponensnek új nevet adunk. Nevezzük át a gombot Foo-ra, álljunk rá a kurzorral, nyomjunk F1-et, és még mindig a TButton osztályról kapunk leírást. Ez azt jelenti, hogy a Delphi a szó összefüggéseit nézi, amikor segítséget kérünk.

Gyakorlatilag két módja van a Súgó előhívásának: válasszuk ki a megfelelő parancsot a **Help** legördülő menüben, vagy válasszuk ki a Delphi környezet egyik elemét illetve egy kódrészletet és nyomjuk meg az F1-et.

Megjegyzendő, hogy Delphiben nem csak egy Súgó fájl van. Legtöbbször, amikor előhívjuk a Delphi súgóját, ki van egészítve egy Object Pascal Help fájllal, Windows API Helppel, a Component Writers Helppel és még sok mással (a Delphi verziójától függően). Ezeknek és a többi Súgó fájlnak egységes külalakja és kereső rendszere van, amit a **Help Topics** gomb megnyomásával aktiválhatunk. A Windows 95 Súgó párbeszédablakának megjelenésekor böngészhetünk a csoportban lévő összes súgófájl tartalomjegyzékében, az Indexben kereshetünk kulcsszó alapján, vagy elindíthatjuk a keresőrendszert. A három lehetőség elérhető a Help Topics párbeszédablak különböző oldalain.

Majdnem mindent megtalálhatunk a Súgóban, de mindenekelőtt tudnunk kell, hogy mit keresünk. Általában ez egyértelmű, de előfordul, hogy nem. Ha egy kis időt fordítunk a Súgó megismerésére, az segít megérteni ezeknek a fájloknak a struktúráját, és megtanulhatjuk, hogy hogyan találjuk meg az információt, amire szükségünk van.

A Súgó fájlok sok információval szolgálnak, mind a kezdő, mind a tapasztalt programozónak, és nagyon értékes referencia eszközként szolgálnak. Tartalmazzák minden egyes komponens összes metódusát és tulajdonságát, az összes metódus vagy funkció paramétereit, és hasonló tudnivalókat, amelyek különösen fontosak lehetnek, amikor programot írunk. A Borland Adobe Acrobat fájlformátumú referencia anyagokat is terjeszt. Ezek az elektronikus verziói a nyomtatott kézikönyveknek, amelyek a Delphi csomaghoz tartoznak, bennük rákereshetünk szavakra, és kinyomtathatjuk a minket érdeklő részeket (vagy akár az egészet is, ha van elég papírunk).

Megjegyzés

A Delphi első verziójában voltak interaktív oktatók (Interactive Tutors). Ha soha nem használtunk Delphit (és fel van installálva a Delphi 1), indítsuk el ezeket az oktatókat. Ezek végigvezetnek bennünket a Delphi alapvető szolgáltatásain, és segítenek megérteni a környezet terminológiáját.

A Delphi Súgó fájlok mellett még sok más tipp és segédanyag is van a Delphi programozók számára. A Delphi CD tartalmaz néhány FAQ (gyakran feltett kérdések) anyagot és a Borland cég technikai információ (Technical Information) gyűjteményét. Mindkét anyag naprakész verzióját megtalálhatjuk

a Borland Web oldalán (http://www.inprise.com). A Borland hivatalos dokumentumai mellett még sok tippet találhatunk a Compuserve fórumokon, és az Internetes levelezési listákon. Nyilvánvaló, hogy a világhálózat komoly információforrás a Delphiről és a külső cégek termékeiről. A CD-n találhatunk egy gyűjteményt ilyen témájú hiperhivatkozásokról (a kedvenc Delphi Web oldalaimat is beleértve).

Tipp

A DTopics a legjobb technikai információ és tippgyűjtemény a Delphiről, melynek hatalmas adatbázisát Mike Orris (3K Computer Consultancy) tartja karban. Ennek az archívumnak saját, speciális megjelenítő programja van. A program és az archívum legújabb verzióját megtalálhatjuk több Internet oldalon is. Szerzői jogi okok miatt ez a gyűjtemény nem található meg a mellékelt CD-n, de annyit mindenképpen elmondhatunk, hogy a DTopics a legjobb Compuserve Forum levelek gyűjteménye a Delphiről.

Delphi menük és parancsok

Alapvetően háromféleképpen adhatunk ki parancsot a Delphi környezetben:

- A menüt használva.
- A SpeedBart (eszköztárat) használva.
- A gyorsmenüt (a jobb egérgomb megnyomására megjelenő menüt) használva.

A Delphi menük sok parancsot ajánlanak fel, azonban nem fogom untatni egy részletes menüstruktúra leírással az Olvasót. Az ilyen jellegű információkat a nyomtatott dokumentációkban vagy a Súgó fájlokban találjuk meg. A következő bekezdésekben bemutatom néhány menüpont használatát, más tanácsok a fejezet hátralevő részében lesznek.

A File menü

Kiindulópontunk legyen a **File** menü. Ez a menü meglehetősen összetett, mert tartalmaz parancsokat, melyek a projekten végeznek műveleteket, és vannak parancsok, amelyek a forráskód fájlokon. Minden alkalommal, amikor megnyitjuk ezt a menüt, kétszer is meg kell gondolnunk, mit is csinálunk. Néhány parancs a projekten és a forráskód fájlokon együtt hajt végre bizonyos műveleteket. A projektekhez kapcsolódó parancsok a **New**, **New Application**,

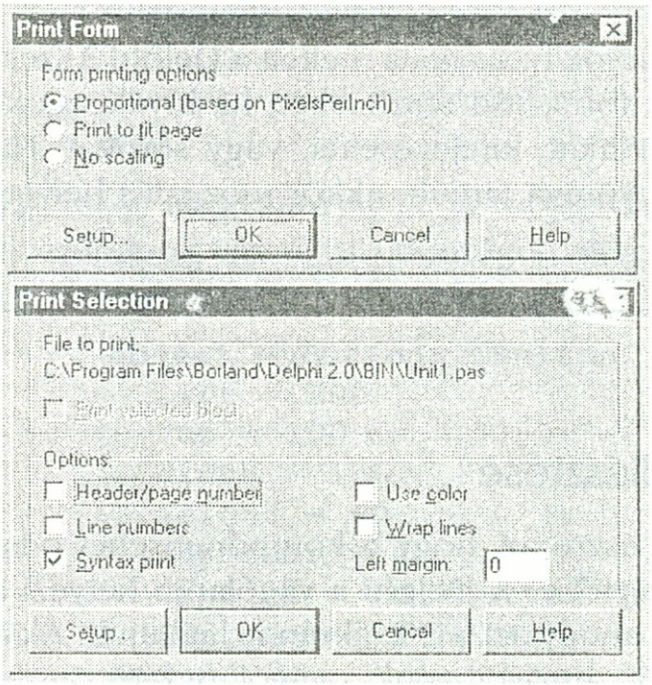
Open, **Reopen**, **Save Project As**, **Save All**, **Close All**, **Add to Project** és **Remove from Project**. Mindezek mellett van egy külön Project legördülő menü is. A forráskód fájlokhoz kapcsolódó parancsok a **New**, **New Form**, **New Data Module**, **Open**, **Reopen**, **Save**, **Save As**, **Close** és **Print**. A legtöbb parancs magáért beszél, de néhánynál nem árt egy kis magyarázat.

Tipp

A **Reopen** menüparancsot olyan projektek vagy forráskód fájlok megnyitására használhatjuk, melyekkel nemrég dolgoztunk.

A **New** parancs gyakorlatilag megnyitja a New Items (új elemek) párbeszédablakot, amelyet Object Repository-nak is hívnak. Ez a párbeszédablak használható arra, hogy előhívjunk vele Delphi varázslókat, vagy új applikációkat, formokat, amelyek már meglévő formokból öröklődnek, szálakat, DLL-eket, Delphi komponenseket és ActiveX kontrolokat hozzunk létre. Az Object Repository sokféle szolgáltatását a következő fejezet tartalmazza.

A másik érdekes parancs a **Print**. Ha éppen forráskódot szerkesztünk, és ezt a parancsot választjuk, akkor a nyomtató a szöveget opcionális szintaxis kiemeléssel nyomtatja ki. Ha éppen egy formon dolgozunk, és akkor választjuk ezt a parancsot, akkor a form grafikus képét nyomtatjuk ki. Természetesen ez



2.1. ábra *A Delphi két különböző nyomtatási ablaka*

hasznos lehet, de össze is zavarhat minket, különösen ha éppen más Delphi ablakokon dolgozunk. Szerencsére két különböző Print párbeszédablak jelenik meg (2.1. ábra), így ellenőrizhetjük, jó műveletet hajtunk-e végre.

Az Edit menü

Az **Edit** menünek van néhány tipikus művelete, mint például az **Undo** és **Redo**, a **Cut**, **Copy** és **Paste** parancsok, és még néhány speciális parancs a form és a szerkesztő ablakok számára. Létezik még egy új Delphi 3 parancs is, az **Add to Interfaces**, ami az ActiveForms fejlesztéséhez kapcsolódik, erről a 23. fejezetben lesz szó.

Fontos megjegyeznünk, hogy az **Edit** menü standard funkciói (és a standard CTRL+Z, CTRL+X, CTRL+C és CTRL+V gyorsbillentyűk) szöveg és form komponenseken egyaránt működnek. Van azonban a megszokott szerkesztési lehetőségekhez képest néhány apróbb eltérés, amit nem árt megemlíteni. Például, amikor a szerkesztőben dolgozunk, a legördülő menü első parancsa az **Undo**, amikor a formon dolgozunk, akkor pedig az **Undelete** lesz. Ennek az az oka, hogy a formszerkesztőnek sajnos nagyon korlátozott a műveletek visszavonására vonatkozó képessége.

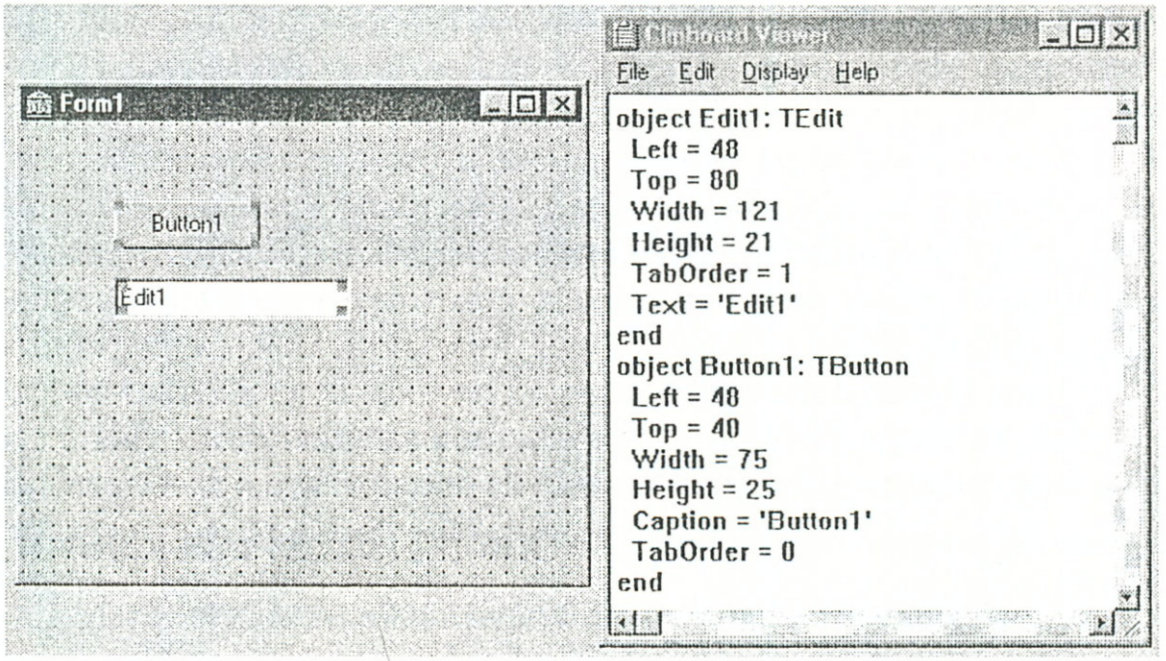
Természetesen kivághatunk és beilleszthetünk szöveget a szerkesztőben, másolhatunk komponenseket formon belül, és egyik formról a másikra is. Akár ugyanannak a formnak egy másik szülő ablakába is beilleszthetünk komponenseket, mint például a panel vagy vezérlőelem-csoport.

Új!

A kivágás és beillesztés parancsok használata mellett a Delphi 3 szerkesztőjében átmozgathatunk forráskód részleteket, ha kiválasztjuk, és az egérrel áthúzzuk a szavakat, kifejezéseket, vagy sorokat. Ha közben a CTRL billentyűt is nyomva tartjuk, akkor mozgatás helyett átmásoljuk a szöveget.

Komponensek másolása és beillesztése

Amit eddig esetleg még nem vettünk észre az, hogy a komponenseket odavissza másolhatjuk a formról a szerkesztőbe. A Delphi a vágólapra helyezi a komponenseket szöveges leírásukkal, ahogy azt a 2.2. ábrán is láthatjuk. Akár át is szerkeszthetjük a szöveges verziót, átmásolhatjuk a vágólapra, és onnan vissza a formra, új komponensként.



2.2. ábra *A komponensek szöveges leírását a Delphi a vágólapra másolja (ahogy azt a Clipboard Viewerben láthatjuk), így bármilyen szövegszerkesztő számára elérhetőek*

Például, ha egy gombot helyezünk a formra, másoljuk, és egy szerkesztőbe beillesztjük (ami lehet a Delphi saját forráskód szerkesztője, vagy bármilyen szövegszerkesztő) akkor a következő leírást kapjuk:

```
object Button1: TButton
  Left = 56
  Top = 48
  Width = 161
  Height = 57
  TabOrder = 0
  Caption = 'Button1'
end
```

Ha most megváltoztatjuk az objektum nevét, címét, vagy pozícióját, vagy hozzáadunk egy új tulajdonságot, akkor ezek a változások visszamásolhatóak egy formra. Lássunk egy példát erre:

```
object MyButton: TButton
  Left = 200
  Top = 200
  Width = 180
  Height = 60
  TabOrder = 0
  Caption = 'My Button'
  Font.Name = 'Arial'
end
```

A fenti leírást a formra beillesztve létrehozunk egy gombot, a megadott pozíción, Arial betűtípusú My Button gombfelirattal. Ahhoz, hogy ez a technika használható legyen, tudnunk kell, hogy kell szerkeszteni egy komponens szöveges változatát, milyen tulajdonságai vannak annak a bizonyos komponensnek, hogy írjuk a szöveges tulajdonságok értékeit, a kapcsoló tulajdonságokat, és más speciális dolgokat. Amikor a Delphi értelmezi egy komponens, vagy form szöveges változatát, akkor megváltoztathatja más tulajdonságok értékeit is, amelyek kapcsolatban állnak az általunk megváltoztatottakkal, és megváltoztatja a komponens formbeli pozícióját, hogy az ne fedje le az előzőt. Mindezt megfigyelhetjük, ha visszamásoljuk a komponenst a szerkesztőbe. Például ezt kapjuk, ha a fenti szöveget a formra másoljuk, és onnan vissza a szerkesztőbe:

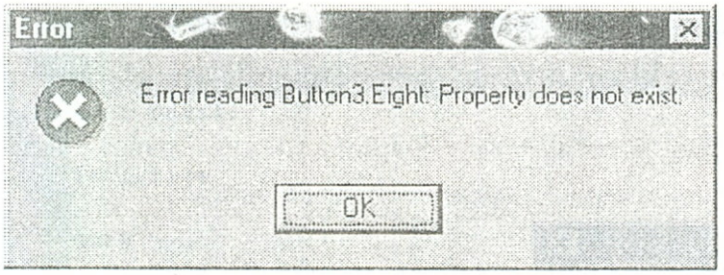
```
object MyButton: TButton
  Left = 112
  Top = 128
  Width = 180
  Height = 60
  Caption = 'My Button'
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Arial'
  Font.Style = []
  ParentFont = False
  TabOrder = 0
end
```

Ahogy láthatjuk, néhány új sor adódott hozzá automatikusan, hogy a betűtípus más tulajdonságait is megadja. Természetesen, ha teljesen rossz sorokat írunk, mint ez a kód:

```
object Button3: TButton
  Left = 100
  eight = 60
end
```

amiben szintaktikai hiba van (egy hiányzó „H" betű), és megpróbáljuk beilleszteni egy formra, akkor egy hibaüzenet jelenik meg, amiből megtudjuk, hogy mit rontottunk el, ahogy azt a 2.3. ábrán is láthatjuk.

Egyszerre akár több komponenst is kiválaszthatunk és másolhatunk egy másik formra, vagy egy szövegszerkesztőbe. Ez hasznos lehet, amikor sok hasonló komponensen kell dolgoznunk: egyet átmásolunk a szerkesztőbe, megsokszorozzuk, elvégezzük a megfelelő módosításokat, és visszamásoljuk az egész csoportot a formra.



2.3. ábra *Egy hibaüzenet látható, amikor megpróbálunk beilleszteni egy komponenst nem létező vagy helytelenül írt tulajdonsággal*

További szerkesztő parancsok

A legtöbb Windows alkalmazás **Edit** menüjében megtalálható tipikus parancsok mellett a Delphi tartalmaz néhány parancsot, amelyek legtöbbször formokkal kapcsolatosak. A jellegzetes, formokra vonatkozókat elérhetjük a gyorsmenüből (lokális menü, amit a jobb egérgomb megnyomásával hívhatunk elő), amelyeket később tárgyalunk a fejezetben.

Megjegyzés

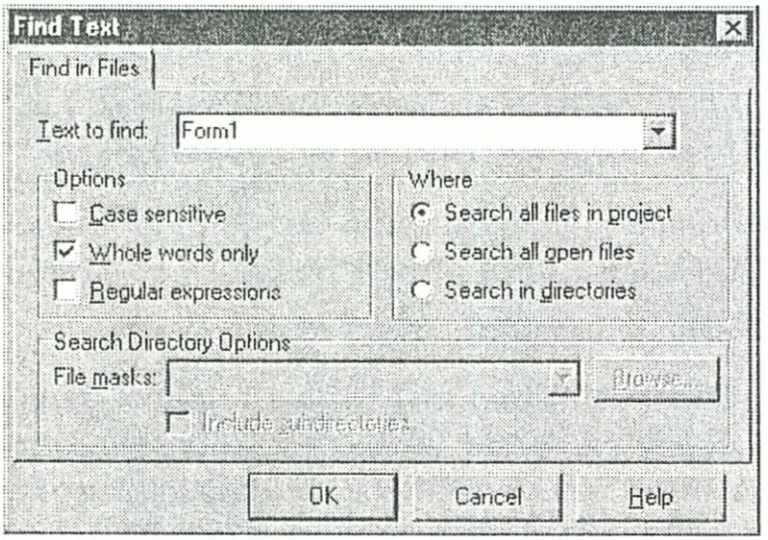
A **Lock Controls** parancs nincs benne a formok lokális gyorsmenüjében, pedig nagyon hasznos, mert megakadályozza a komponensek pozíciójának véletlenszerű megváltoztatását a formon. Jó példa erre az, mikor megpróbálunk duplán kattintani egy komponensre, és véletlenül feljebb mozdítjuk. Mivel nincs Undo művelet a formoknál, az ilyen hibák ily módú megelőzése nagyon hasznos lehet egy már teljesen megtervezett form esetében.

A Search menü

A **Search** (Keresés) menünek is van néhány standard parancsa, mint például a **Search and Replace** (Keresés és Csere), és a Delphi 3 új **Find in Files** (Keresés fájlokban) parancsa (a 2.4. ábrán láthatjuk a parancs párbeszédablakát).

A **Find in Files** paranccsal a projekt összes forráskód fájljában, az összes megnyitott fájlban, vagy egy könyvtár összes fájljában (opcionálisan a könyvtár alkönyvtáraiban is) kereshetünk egy szöveget, attól függően, hogy melyik rádiógombot választjuk. A keresés eredménye egy üzenetablakban jelenik

meg, a szerkesztő panel alsó felében. Kiválaszthatunk egy bejegyzést, ha meg akarunk nyitni egy fájlt a listából, ekkor arra a sorra ugrunk majd, ami a szöveget tartalmazza.



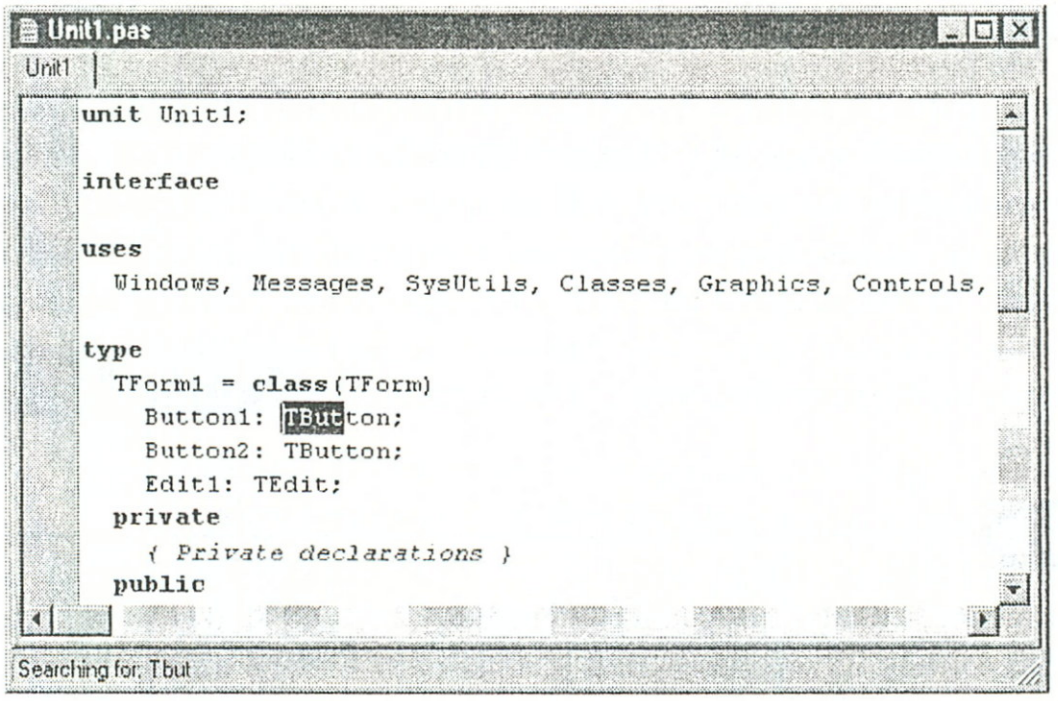
2.4. ábra *Az új Find in Files párbeszédablak hasznos lehet adott szöveg több fájlban való keresésére*

Megjegyzés

A **Find in Files** paranccsal komponens, osztály és típus definíciókat kereshetünk a VCL (Visual Component Library) forráskódjában (ha telepítettük azt). Ez egy jó módszer részletes információk szerzésére egy komponensről, ámbár a Súgó használata általában gyorsabb és egyszerűbb.

Más parancsokat nem olyan egyszerű megérteni. Az **Incremental Search** (növekményes keresés) az egyik. Amikor ezt a parancsot választjuk, ahelyett, hogy egy párbeszédablak jelenne meg, ahol beírhatjuk a keresendő szöveget, a Delphi a szerkesztő ablakra pozícionál. Ott, közvetlenül az üzenet ablakba írhatjuk be a keresendő szöveget, ahogy azt a 2.5. ábrán is láthatjuk.

Amikor beírjuk az első betűt, a szerkesztő az első olyan szóra pozícionál, amelyik a beírt betűvel kezdődik (ha a keresett szöveget nem találta meg, akkor a betűk meg sem jelennek a szerkesztő üzenet ablakában). Ha ez nem az a szó, amit kerestünk, akkor csak folytassuk a gépelést, a kurzor folytatja az ugrást, ahogy egyre több betűt írunk be. Habár ez a parancs furcsának tűnhet első



2.5. ábra *Az Incremental Search parancs használatának példája*

látásra, nagyon hatékony és rendkívül gyors, különösen akkor, ha éppen gépeltünk, és egy gyorsbillentyűvel aktivizáljuk (ami CTRL+E, ha az alap gyorsbillentyű beállításokat használjuk).

A **Go to Line Number** (adott sorra ugrás) és a **Show Last Compile Error** (utolsó fordítási hiba megjelenítése) parancsok eléggé egyértelműek. A **Find Error** parancs furcsának tűnhet első látásra. Ezt adott futásidejű hiba keresésére használjuk, nem pedig fordítási hiba keresésére. Amikor egy önálló programot futtatunk, és egy végzetes hibába botlunk, a Delphi kiír egy belső hivatkozási címet (ami a lefordított kód logikai címe). Beírhatjuk ezt a számot a Find Error párbeszéd ablakba, hogy a Delphi lefordítsa újra a programot a megadott értéket keresve; amikor megtalálta a címet, akkor kiírja a megfelelő forráskód sort. Gyakran előfordul, hogy a hiba nem a mi programunkban van, hanem egy könyvtár vagy rendszer kódsorában: ebben a meglehetősen gyakori esetben a **Find Error** parancs nem találja meg a rossz sort.

A **Search** menü utolsó parancsa a **Browse Symbol**, ami előhívja az Object Browser-t (objektum böngésző), amit a lefordított programban definiált szimbólumok felkutatására használhatunk. Az Object Browser működésének megértéséhez szükséges az Object Pascal nyelv és a Visual Component Library (VCL) alapos ismerete. Az Object Browsert a 26. fejezetben fogjuk használni.

A View menü

A **View** (Nézet) legördülő menü a **View** és a **Window** menükben általában megtalálható funkciókat kombinálja. Nincsen **Window** (Ablak) menü, mert a Delphiben egyszerre csak egy projektünk lehet nyitva. A **View** menü legtöbb parancsa a Delphi egy ablakának megjelenítésére használható, mint például a **Project Manager**, **Breakpoints List** vagy a **Components**. Néhány ablakot ezek közül hibakeresés alatt használunk, másokat pedig programírás közben. A legtöbb ilyen ablakról beszélünk még a fejezet hátralevő részében.

Megjegyzés

A **View** menühöz lehetséges egy új menüpont, a **CPU Window** hozzáadása, amely hibakeresés alatt használható. Ebben az ablakban a Delphi fordítója által generált assembley kódok lépésenkénti végrehajtása közben láthatjuk a CPU regisztereinek állapotát. Többet is megtudhatunk erről az ablakról és használatáról a 26. fejezetben.

A **View** menü második részében található parancsok nagyon fontosak, ezért a SpeedBar-on szintén el lehet őket érni (a SpeedBar a főmenü alatt található eszköztár, amely a legfontosabb Delphi parancsokat, a mentést, megnyitást, futtatást, stb. tartalmazza). A **Toggle Form/Unit** (vagy F12) paranccsal a form és a hozzátartozó forráskód között válthatunk. Ha nagy méretű forráskódszerkesztő ablakot használunk, akkor gyakran fogjuk használni ezt a parancsot. Alternatívaként a két ablakot (a szerkesztőt és a formot) elhelyezhetjük úgy is, hogy az alul lévőnek mindig legyen látható része. Ilyen elrendezéssel csak rá kell kattintani az egérrel arra, amit meg szeretnénk jeleníteni.

A **New Edit Window** parancs megnyit egy második szerkesztő ablakot. Ez az egyetlen módja Delphiben, hogy két fájlt egymás mellett nézhessünk, mivel a szerkesztő „füles" oldalakat használ több fájl megjelenítésére. Miután megduplázuk a szerkesztő ablakot, mindkettőben különböző fájlokat, vagy ugyanannak a fájlnak két különböző részét nézhetjük.

A **View** menü utolsó két parancsát a SpeedBar vagy a komponenspaletta elrejtésére használhatjuk, ezt azonban nem ajánlom, mert így kényelmetlen kezelni a Delphit. Formokon dolgozni a komponenspaletta nélkül meglehetősen nehéz. Ha a SpeedBart és a komponens palettát is elrejtjük, akkor a Delphi fő ablakát egy csupasz menüre redukáljuk.

A Project menü

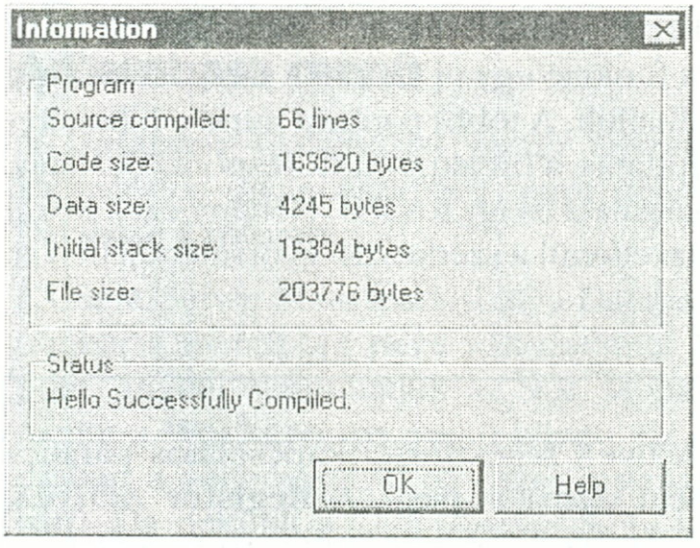
A következő legördülő menü a **Project**, melynek parancsai a projektet lefordítják, menedzselik. Az **Add to Project** és **Remove from Project** parancsokkal formokat és Pascal forráskód fájlokat adhatunk hozzá, vagy vehetünk ki a projektből.

Új!

A Delphi Client/Server verziójában van még két parancs, a **Web Deploy Options** és a **Web Deploy**, melyek nem találhatóak meg a többi verzióban. Az utolsó fejezet végén, az ActiveFormok létrehozásáról szóló részben majd képet kaphatunk ezeknek a parancsoknak a szerepéről.

A **Compile** parancs le- , vagy újrafordítja az applikációt, futtatható fájlt hozva létre, ellenőrzi, hogy a forráskód fájlok voltak-e módosítva, és ha szükséges, akkor újrafordítja őket. A **Build All** paranccsal a projekthez tartozó összes forráskód fájlt lefordíthatjuk, ami akkor is újrafordítja az összes fájlt, ha azokat az utolsó fordítás óta nem módosítottuk. Ha a kódot csak szintaktikailag akarjuk ellenőrizni, futtatható fájl létrehozása nélkül, akkor használjuk a **Syntax Check** parancsot.

A következő parancs az **Information**, az utolsó fordításról jelenít meg részleteket. A 2.6. ábrán láthatjuk az 1. fejezetben bemutatott program (Hello2) fordítási részleteit.



2.6. ábra *Információk az előző fejezet Hello2 példaprogramjának fordításáról*

Megjegyzés

A **Compile** parancsot csak akkor használhatjuk, ha már betöltöttünk egy projektet a szerkesztőbe. Ha nincs aktív projektünk, és betöltünk egy Pascal forráskódot, nem tudjuk lefordítani, de ha a forráskódot projektként töltjük be, akkor ezzel a trükkel lefordíthatjuk azt. Egyszerűen csak nyomjuk meg az **Open** gombot az eszköztáron, és töltsünk be egy PAS fájlt. Most ellenőrizhetjük a szintaktikai helyességét, vagy lefordíthatjuk, egy DCU fájlt létrehozva (Delphi Compiled Unit).

A **Project** menü utolsó menüpontja az **Options**, ami a fordító és a linker, valamint az alkalmazás-objektum opcióinak, és még sok más beállítására szolgál. Ha megváltoztatjuk a projekt opciókat, akkor a **Default** jelölőnégyzet kitöltésével megadhatjuk, hogy ugyanezekkel a beállításokkal hozza létre az új projekteket is. A projekt opciókat ebben a fejezetben, és a könyv e témához kapcsolódó részeiben még újra megbeszéljük.

A Run menü

A **Run** (Futtatás) menüt akár Debug (Hibakeresés) menünek is nevezhették volna. A legtöbb parancsa a hibakereséshez kapcsolódik, a **Run** parancsot is beleértve. Amikor a Delphi környezetben futtatunk egy programot, akkor egy beépített debugger alatt futtatjuk (hacsak nem kapcsoltuk ki a megfelelő környezeti paraméterben). A **Run** parancs, és a hozzátartozó SpeedBar ikon a leggyakrabban használt parancs, mivel a Delphi automatikusan lefordítja a programot mielőtt futtatná – legalább akkor, ha a forráskód megváltozott. Ha inkább a billentyűzetet szeretjük használni, egyszerűen nyomjuk meg az F9-et (mint gyorsbillentyűt) a program fordításához és futtatásához.

A **Parameters** parancsot a futtatandó program parancssori paraméterek megadására, és DLL debuggolásakor (ez a funkció egy új Delphi 3 szolgáltatás) egy futtatható fájlnév megadására használhatjuk. A többi parancs mind a hibakeresés alatt használatos, a lépésenkénti futtatás, a breakpoint (töréspont) beállítás, objektumok és változók értékének vizsgálata és így tovább. Néhány ezek közül a debug parancsok közül elérhető közvetlenül a szerkesztő gyorsmenüjében is.

Új!

A Delphi 3-ban több, az ActiveX fejlesztéssel kapcsolatos parancs van (a Standard kiadásban nem találhatók meg). A **Register ActiveX Server** és az **Unregister ActiveX Server** menüparancsok gyakorlatilag a jelenlegi projekt által definiált ActiveX-el kapcsolatban adnak hozzá, vagy törölnek információkat a Windows Registry-ben.

A Component menü

A **Component** (Komponens) menü parancsai komponensek írására, csomagokhoz való hozzáadására, vagy csomagok Delphiben való installálására használhatók. A **New Component** (Új komponens) parancs egyszerűen előhívja a Component varázslót. A három installáló parancs, **Install Component**, **Import ActiveX Library** és **Install Packages** arra használható, hogy a környezethez új Delphi komponenseket, csomagokat vagy ActiveX-et adjunk hozzá. A három parancs közül bármelyiket elindítva a megfelelő csomaghoz és komponens palettához új elemeket adhatunk hozzá. Ezeket a parancsokat részletesen megbeszéljük a 6., 18. és a 23. fejezetekben.

Komponens sablonok

Az előző fejezetben röviden használtuk a **Create Component Template** menüpontot. Miután kijelöltünk egy vagy több komponenst a formon és kiadjuk ezt a parancsot, megjelenik egy párbeszédablak (ahogy azt az első fejezetben már láthattuk), ahol választhatunk egy ikont és lapot a palettában, és megadhatjuk a nevét az új komponens sablonnak. Alapértelmezés szerint a sablon neve az első kiválasztott komponens neve, amihez a Delphi a template szót hozzáfűzi. Az ikon is alapértelmezés szerint az első kiválasztott komponens ikonja, amit azonban kicserélhetünk más ikonfájlra. A komponens paletta a megadott nevet használja a komponens sablon leírására (amikor a Delphi megjeleníti az ide vonatkozó tippet).

A komponens sablonok összes információja a DELPHI32.DCT fájlban van tárolva, de úgy tűnik, nincs mód az ebben tárolt információk visszanyerésére, és egy sablon szerkesztésére. Annyit tehetünk, hogy helyezzünk egy komponens sablont egy teljesen új formra, szerkesszük át, és installáljuk újra ugyanazon a néven. Ilyen módon fölül tudjuk írni az előzőleg definiált sablont.

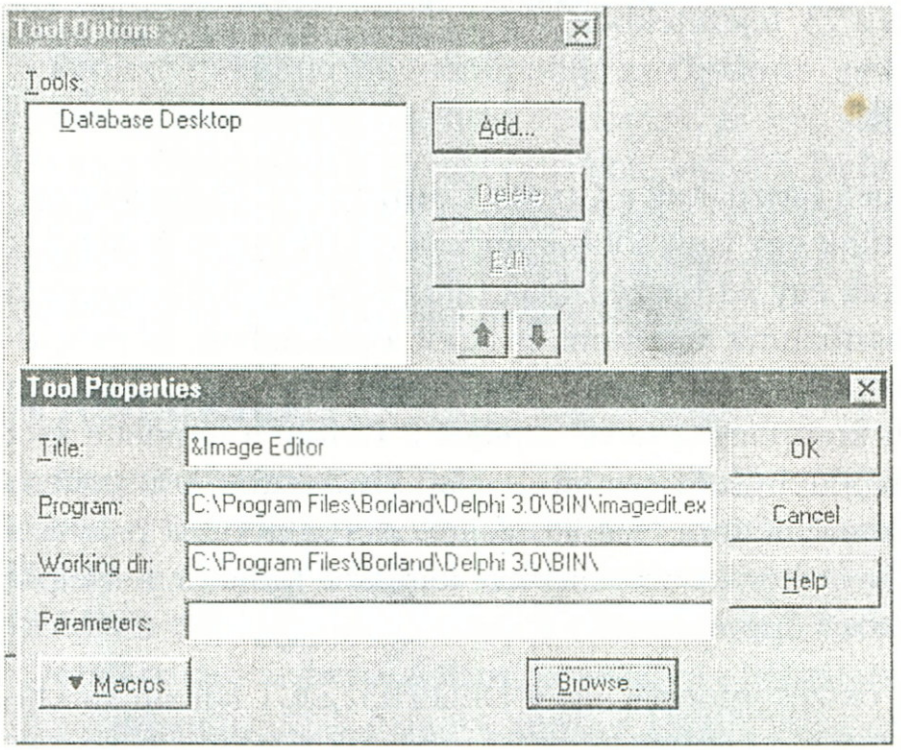
A Database menü

A **Database** (Adatbázis) menü összefogja a Delphi adatbázis-kezeléssel kapcsolatos olyan eszközeit, mint például a Database Form Wizardot és a Database Explorert.

A Client/Server Suite kiadásban a Database Explorer helyett SQL Explorer van (ámbár a menüpont neve változatlanul **Database Explorer**), és egy menüpont, amivel elindíthatjuk az SQL Monitort. Ebben a kiadásban a **Database** menüt a **Workgroups** menü követi. Ez a legördülő menü a Professional kiadásban a verziővezérlő rendszer installálásával aktiválható (nincs a csomagban). Ha nekünk a Client/Server Suite kiadás van meg, és nem használjuk a hozzácsatolt VCS-t, akkor azt el kell távolítani úgy, hogy a megfelelő bejegyzéseket a Windows Registryből ki kell törölni (a Software/Borland/Delphi3.0 részben).

A Tools menü

A **Tools** menü egyszerűen egy pár külső program és eszköz listája, amely megkönnyíti az elindításukat. A **Tools** parancsot a **Tools** menü konfigurálására, és új külső programok hozzáadására használhatjuk. Ahogy azt a 2.7. ábrán is láthatjuk, a program egyszerű elindítása mellett még paramétereket is átadhatunk. Egyszerű paraméterlistákat közvetlenül a parancssorban adhatunk át, míg összetettebbeket a **Tool Properties** párbeszédablak alsó felében található **Macros** gomb megnyomásával szerkeszthetünk.



2.7. ábra *Egyszerűen lehet új alkalmazásokat a Tools menühöz hozzáadni*

A **Tools** menü tartalmaz egy parancsot, amivel a Repository-t (melyről a következő fejezetben lesz szó), és az **Options** parancsot, amellyel az egész Delphi fejlesztői környezetet konfigurálhatjuk. Az Environment Options párbeszédablaknak vannak lapjai, amelyek különféle környezeti beállításokhoz kapcsolódnak (a **Preferences** lap a 2.9. ábrán később látható), csomag és könyvtár beállításokhoz, számos szerkesztési opcióhoz (az **Editor**, **Display** és a **Color** oldalak), van egy lap, amin a komponens palettát konfigurálhatjuk, egy az Object Browsernek, és egy az új Code Insight technológiának. Sok opciót megbeszélünk majd a kapcsolódó szolgáltatások feltárásakor.

A Help menü

A Help menüben információkat szerezhetünk a Delphiről (**Help** menü **Help Topics** menüpontja), és a Delphi névjegy ablakát is megjeleníthetjük. Ebben az ablakban, ha nyomva tartjuk az ALT billentyűt, és begépeljük a VERSION szót, akkor láthatjuk a Delphi verzió számát. Más billentyű-kombinációk esetén (TEAM) azoknak az embereknek a listáját láthatjuk, akik részt vettek a Delphi elkészítésében. A **Help** menüben gyakran külső cégek által készített Delphi varázslók vannak (vagy Experts – szakértők –, ahogy azokat a Delphi 2-ben nevezték), bár némelyik varázsló más legördülő menühöz is hozzáad menüpontokat.

A Delphi eszköztára

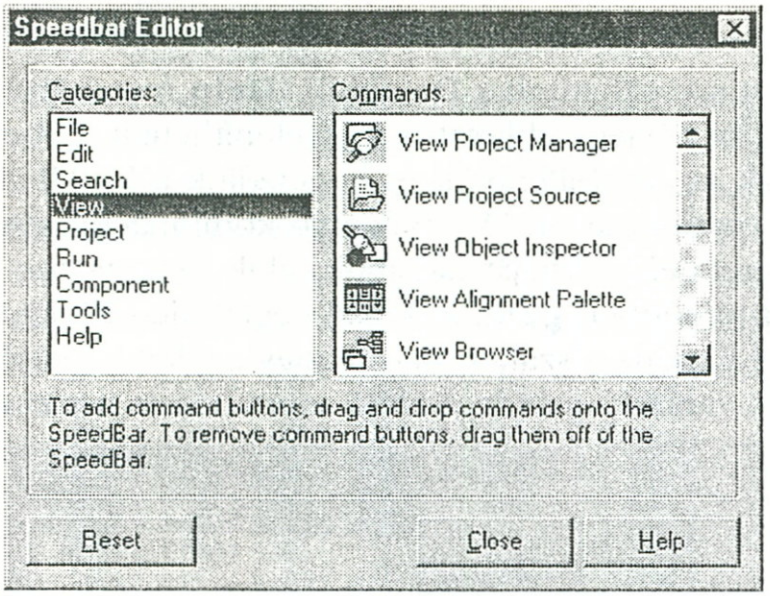
Ha már egy ideje használjuk a Delphit, akkor láthatjuk, hogy a parancsok csak egy kis részét használjuk gyakran. Néhány ezek közül már valószínűleg a SpeedBar-on van (az eszköztár elnevezése a Borlandnál), néhány pedig nincs. Ha a parancsok, amelyeket sokat használunk nincsenek rajta, akkor itt az ideje, hogy beállítsuk a SpeedBart, hogy hatékonyabb segítséget nyújtson a Delphi használatában.

Tipp

A SpeedBar használata mellett egy másik lehetőség a billentyűzet-parancsok használata. Habár emlékeznünk kell néhány billentyű-kombinációra, hogy használhassuk őket, e parancsokkal nagyon gyorsan adhatunk ki utasításokat, különösen ha éppen programot írunk, és az ujjaink már amúgy is a billentyűzeten vannak.

Könnyedén átméretezhetjük a SpeedBar-t, ha a vékony vonalat mozgatjuk közte és a komponenspaletta között. De a legfontosabb művelet amit a SpeedBarral tehetünk az, hogy hozzáadunk, elveszünk, vagy kicseréljük az ikonokat a gyorsmenüje **Configure** parancsával (egyszerűen csak nyomjuk meg a jobb egérgombot rajta). Ez a parancs előhívja a SpeedBar Editor ablakot (2.8. ábra), ami talán az egyik legjobban kezelhető felületű Delphi eszköz.

Egy ikont úgy adhatunk hozzá a SpeedBarhoz, hogy megfelelő kategória mellett megtaláljuk (hasonlóan egy legördülő menühöz), és átmozgatjuk a SpeedBarra. Ugyanezen a módon kivehetünk egy ikont a SpeedBarról, vagy egyszerűen csak átmozgatjuk egy másik helyre. A könnyebb kezelhetőség kedvéért ikon csoportokat is kialakíthatunk, ha a csoportok között hagyunk egy kis helyet.



2.8. ábra *A SpeedBar Editor. Ha egyszer már nyitva van, akkor az egérrel áthúzhatunk ikonokat róla a SpeedBarra*

A lokális menük

Habár a Delphinek elég sok menüpontja van, nem érhető el az összes parancs a legördülő menükön keresztül. Néha használnunk kell a lokális menüket, amelyeket gyorsmenünek is hívnak a Borland zsargonjában, bizonyos ablakokon, vagy ablakrészeken. A gyorsmenü aktiválásához nyomjuk meg a jobb egérgombot az ablakon, vagy nyomjuk meg az ALT+F10-et. Még ha más lehetőségeink is vannak, a gyorsmenü használata általában gyorsabb, mert nem kell az egérrel a főmenüben keresgélnünk. Gyakran egyszerűbb is, mivel a gyorsmenü parancsai az aktuális ablakhoz kapcsolódnak. A Delphi majdnem összes ablakának (a párbeszédablakok kivételével) megvan a saját gyorsmenüje, a rá vonatkozó parancsokkal. Tényleg ajánlom, hogy szokjunk hozzá az ablakok gyorsmenüjének használatához, mert nem csak a Delphiben fontos ez, hanem az összes Windows 95 alkalmazásban. Szokjuk meg, és az általunk készített Delphi applikációkba is építsük bele ezt a lehetőséget.

A Form Designer használata

A Delphi környezetben a formok tervezése az alapja a vizuális fejlesztésnek. Minden komponens, amit a formra helyezünk, és minden tulajdonságot, amit beállítunk, egy, a formot leíró fájlban van letárolva (egy DFM fájl), aminek a formhoz tartozó forráskód fájlra (a PAS fájl) is van hatása.

Amikor egy új, üres projektet indítunk, akkor a Delphi egy üres formot hoz létre, amivel elkezdhetünk dolgozni. Egy már meglévő formmal is kezdhetünk (a különféle sablonokat használva), vagy új formokat adhatunk a projekthez. Egy

projektnek (alkalmazásnak) akárhány formja lehet. Minden alkalommal, amikor egy formon dolgozunk, akkor gyakorlatilag a Delphi Form Designer-ét használjuk. Munka közben állíthatjuk a form, vagy egy komponensének a tulajdonságait, vagy egyszerre több komponenssel is végezhetünk műveleteket. A form, vagy annak egy komponensének kiválasztásához egyszerűen kattintsunk rá, vagy használjuk az objektum kiválasztót (Object Selectort – az Object Inspector ablakban található lista), ahol mindig láthatjuk a kiválasztott elem nevét és típusát. Egyszerre több komponenst is kiválaszthatunk, ha a SHIFT-et lenyomva kattintunk a komponensekre, vagy ha egy kiválasztó téglalapot húzunk a komponensek körül a formon.

Tipp

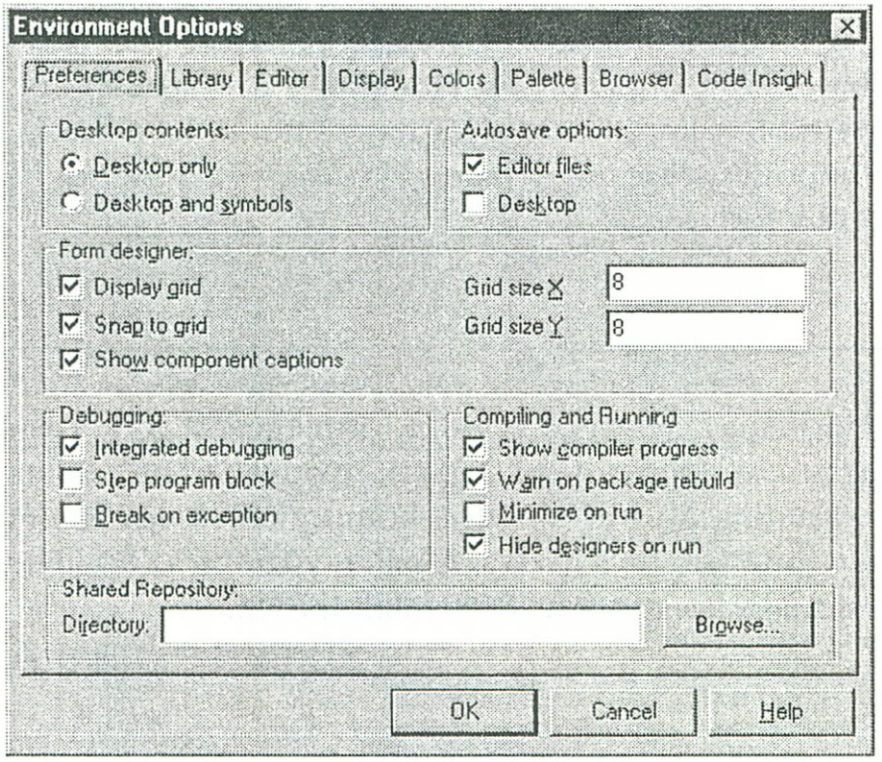
Még ha egy komponens teljesen be is fedi a form egészét, akkor is ki tudjuk azt választani az egérrel. Csak tartsuk nyomva a SHIFT billentyűt, miközben rákattintunk a kiválasztott komponensre. Így alapértelmezés szerint a form lesz kiválasztva. A billentyűzeten az Esc megnyomásával az aktuális komponens ősét választjuk ki.

Amíg a formon dolgozunk, a gyorsmenünek van néhány hasznos parancsa (néhány az **Edit** menüben is megtalálható). A **Bring to Front** és a **Send to Back** parancsokat az ugyanolyan típusú komponensek relatív helyzetének megváltoztatására használhatjuk (egy grafikus komponenst soha sem hozhatunk egy ablakon található komponens elé, ahogy azt a 6. fejezetben látni fogjuk). Egy örökített formon használhatjuk a **Revert to Inherited** parancsot, hogy visszaállítsuk a kiválasztott komponens tulajdonságait a szülő értékeire.

Amikor egynél több komponenst választunk ki, akkor igazíthatjuk, és méretezhetjük is őket. Az Alignment (Elrendezés) párbeszédablak legtöbb opciója megtalálható az Alignment palettában is (amit a **View** menü **Alignment Palette** parancsán keresztül érhetünk el). A Tab Order és a Creation Order párbeszédablakokat is megnyithatjuk, hogy be tudjuk állítani a látható komponensek, kontrolok tabulátor sorrendjét, és a nem látható kontrol objektumok létrehozási sorrendjét. Az **Add to Repository** paranccsal a formot, amin dolgozunk hozzáadhatjuk a más projektekben felhasználható formok listájához. Végül a **View as Text** paranccsal a formot bezárjuk, és megnyitjuk egy szerkesztőben a szöveges leírását. A megfelelő parancs a szerkesztő gyorsmenüjében (**View as Form**) megfordítja az előző eljárást.

A gyorsmenü parancsai mellett a **Tools** menü **Options** parancsával beállíthatunk néhány form opciót a **Preferences** oldalon (lásd 2.9. ábra). A formmal kapcsolatos opciók a Form Designer fejléc alatt vannak kilistázva, és a rács

bekapcsolására és méretére vonatkoznak. A rács megkönnyíti a komponensek formon való elhelyezését úgy, hogy adott rácspontokhoz tudjuk igazítani és méretezni a komponenseket. Rács használata nélkül igen körülményes két komponens egérrel való, manuális igazítása.



2.9. ábra *Az Environment Options párbeszédablak*

Van két alternatíva egy komponens egér nélküli elhelyezésre: beállíthatjuk a Left és a Top tulajdonságokat, vagy a CTRL billentyűt lenyomva tartva a kurzorbillentyűkkel a helyére vihetjük a komponenst. E billentyűk használata különösen hasznos lehet egy elem helyzetének finom beállítására (a **Snap to Grid** opció csak egérműveletekre vonatkozik). A SHIFT billentyűt nyomva tartva a komponens finom méretezését is el tudjuk végezni. Ha a SHIFT+CTRL billentyűk nyomva tartása mellett nyomjuk le a kurzorbillentyűket, a komponens csak rácspontokon mozog.

Az eddig ismertetett parancsok mellett egy form gyorsmenüje néhány komponens kiválasztásakor más parancsokat is felajánl. Bizonyos esetekben ezek a menüparancsok a komponens tulajdonságaira vonatkoznak, és más különlegesen hasznos parancsokat is tartalmazhatnak. A 2.1. táblázat tartalmazza a parancsok listáját, ami a formon található komponensek kiválasztásakor a gyorsmenühöz adódnak (a TeeChart, Quick Report és Decision Cube komponenseket nem tartalmazza a lista, mert nagyon sok parancsot adnak a

gyorsmenühöz). Megjegyzendő, hogy bizonyos esetekben ezek a parancsok a komponensek alapfunkciói is, amelyek, ha a Form Designer-ben duplán kattintunk a komponensre, automatikusan aktiválódnak.

2.1. táblázat

Gyorsmenühöz hozzáadott parancsok különböző komponensek kiválasztásakor

Menüparancs	Komponensek
Menu Designer	MainMenu, PopupMenu
Query Builder	Query (ha a Visual Query Builder elérhető)
Fields Editor	Table, Query, StoredProc, ClientDataSet
Explore	Table, Query, StoredProc, Database (ha a Database Explorer vagy az SQL Explorer elérhető)
Define Parameters	Query, StoredProc
Database Editor	Database
Assign Local Data	ClientDataSet
Update SQL Editor	UpdateSQL
Execute	BatchMove
Columns Editor	DBGrid
Edit Report	Report
ImageList Editor	Image List
New Page	Page Control
Next Page, Previous Page	Page Control, NoteBook, TabbedNoteBook
Next Frame, Previous Frame	Animate
Insert Object	OLE Container
New Button, New Separator	ToolBar
Properties, About	ActiveX kontrolok
ResponseEditor	QueryTableProducer, DataSetTableProducer

A komponenspaletta

Amikor egy új komponenst akarunk a formhoz adni, akkor a komponenspaletta egyik oldalán rá kell kattintanunk egy komponensre és utána a formra. A formon a bal egérgombot nyomva tartva és az egeret mozgatva beállíthatjuk a komponens méretét és helyét, vagy csak kattintsunk egyet, ha az alapértelmezés szerinti méretet szeretnénk használni.

A paletta minden egyes lapján van néhány komponens; mindegyiknek van ikonja és neve, ami a szálló-tippben jelenik meg (ha az egérrel az ikonjára állunk, és várunk egy másodpercet). A tippek a komponensek hivatalos elnevezését írják ki, amelyet a könyvben is használunk. A komponensek neve az őket definiáló osztály nevéből ered, a kezdő T betű nélkül (például, ha az osztály a TButton, akkor a komponens neve Button).

Tipp

Ha egy formra több, ugyanolyan típusú komponenst kell helyeznünk, akkor a palettában a SHIFT-et nyomva tartva válasszuk ki a komponenst. Ezután minden alkalommal, ha a formra kattintunk a Delphi egy új, ugyanolyan típusú komponenst tesz a formra. Ezt úgy állíthatjuk meg, hogy a komponens paletta Standard lapjának bal szélén található nyíl ikonra kattintunk.

Ha esetleg egér nélküli gépet használunk, és egy komponenst akarunk a formhoz adni, akkor a **View** menü **Components List** parancsával megtehetjük. Válasszunk ki egy komponenst a megjelenő listából, vagy írjuk be a nevét, és kattintsunk az **Add to Form** gombra.

Természetesen teljesen átrendezhetjük a komponenseket a paletta különböző oldalait, új elemeket hozzáadva, vagy a komponensek egyik lapról a másik lapra mozgatásával: válasszuk ki a **Tools** menü **Options** menüpontját, és a **Palette** oldalt. A párbeszéd ablaknak ezen az oldalán egyszerűen megfogunk az egérrel egy komponenst a **Component** listából, és a **Pages** listába húzzuk. Nem túl jó ötlet gyakran mozgatni a komponenseket a palettában, mert az állandó keresgéléssel jár, ami időpazarlás.

Tipp

Ha túl sok lapunk van a komponens palettában, akkor görgetnünk kell őket, ha el akarunk érni egy komponenst. Van egy egyszerű trükk, amit ilyen esetben használhatunk: nevezzük át az oldalakat rövidebb nevekre, hogy az összes lap elférjen a képernyőn.

Az Object Inspector

Amikor a formot tervezzük, akkor az Object Inspectort (Objektum Vizsgáló) használjuk a komponensek és a form tulajdonságainak beállítására. Az Object Inspector ablakában vannak kilistázva a kiválasztott elem tulajdonságai (és eseményei), és a beállított értékek két átméretezhető oszlopban. Az ablak felső részében van az Object Selector (Objektum Kiválasztó), ami az éppen aktuális komponens nevét és típusát jelzi, és itt is beállíthatjuk, hogy melyik legyen a kiválasztott komponens. Az Object Inspector nem az összes tulajdonságát listázza ki a komponenseknek, csak azokat tartalmazza, amelyek a szerkesztés

során beállíthatóak. Ahogy azt az 1. fejezetben már említettük, más tulajdonságok csak futásidő alatt érhetőek el. Az egyes komponensek összes tulajdonságát a súgófájlokban találjuk meg.

Az Object Inspector jobb oldali oszlopában állíthatjuk be a tulajdonságok értékeit (csak megfelelő típusú értékeket vihetünk be). A tulajdonság típusától függően stringet, vagy számokat adhatunk meg, listából választhatunk (ezt egy nyíl jelzi), vagy egy bizonyos szerkesztőt hívhatunk elő (ezt három pont jelzi). Amikor egy tulajdonság csak két értéket engedélyez, mint például True, vagy False, akkor az értékeket váltogathatjuk, ha duplán kattintunk rajtuk. Ha több érték is van, akkor a dupla kattintás mindig a következő értéket választja ki a listából. Ha sokszor ismételjük ezt, akkor a lista összes értéke sorra megjelenik, de egyszerűbb kiválasztani egy értéket a többelemű listából. Néhány tulajdonságnál, mint például a Color (Szín), beírhatunk egy értéket, választhatunk listából, vagy akár egy szerkesztőt is előhívhatunk! Más tulajdonság, mint például a Font, beállítható az altulajdonságainak kibontásával (egy plusz, vagy mínusz jel jelzi a neve mellett), vagy egy szerkesztő előhívásával. Más esetekben, mint például a string listák, a speciális szerkesztő az egyetlen módja a tulajdonság beállításának. Az altulajdonság mechanizmus halmazoknál és osztályoknál érhető el. A 4. fejezetben lesz szó a halmazokról és a működésükről, az Object Pascal osztályokról pedig az 5. fejezetben beszélünk. Amikor az Object Inspectorban az altulajdonságokat kibontjuk, mindegyiknek megvan a saját „viselkedése", ismételten a típusától függően.

Az Object Inspectort sűrűn fogjuk használni, ezért jó, ha mindig látható, amikor egy formot szerkesztünk, de ha programot írunk, akkor sem árt látni a komponensek nevét, és tulajdonságait. Ezért ennek az ablaknak a gyorsmenüjében van egy **Stay on Top** nevű parancs, ami mindig a form és a szerkesztő előtt tartja az Object Inspectort.

Az Alignment paletta

Az utolsó eszköz, ami a form szerkesztéséhez kapcsolódik, az Alignment paletta. Ezt a **View** menü **Alignment palette** parancsával nyithatjuk meg. Egy másik lehetőség, hogy kiválasztjuk azokat a komponenseket, amelyeket igazítani akarunk, és kiadjuk a form gyorsmenüjének **Align** parancsát.

Az Alignment paletta funkciói közé tartozik, a sokféle kontrol pozícionálása, középre helyezése, egymástól való távolságuk beállítása, és így tovább. A gombok funkciójának megállapításához egyszerűen csak mozgassuk az egeret föléjük, és a szálló-tippekben megjelenik jelentésük. Amikor összetett formokat szerkesztünk, célszerű az Alignment palettát is mindig látható pozícióba mozgatni, vagy gyorsmenüjének **Stay on Top** parancsát használni.

Programkód írása a szerkesztőben

Miután megszerkesztettük a formot, általában szükséges néhány eseményére programot írnunk, ahogy azt az 1. fejezetben is tettük. Minden alkalommal, amikor egy eseményen dolgozunk, a Delphi megnyit egy szerkesztőt, amiben a formhoz tartozó forráskód fájlt szerkeszthetjük. Könnyedén váltogathatunk a form szerkesztő, és a forráskód szerkesztő között a SpeedBar **Toggle Form/Unit** gombjára kattintva, a megfelelő ablakra kattintva, vagy az F12 funkcióbillentyűt megnyomva.

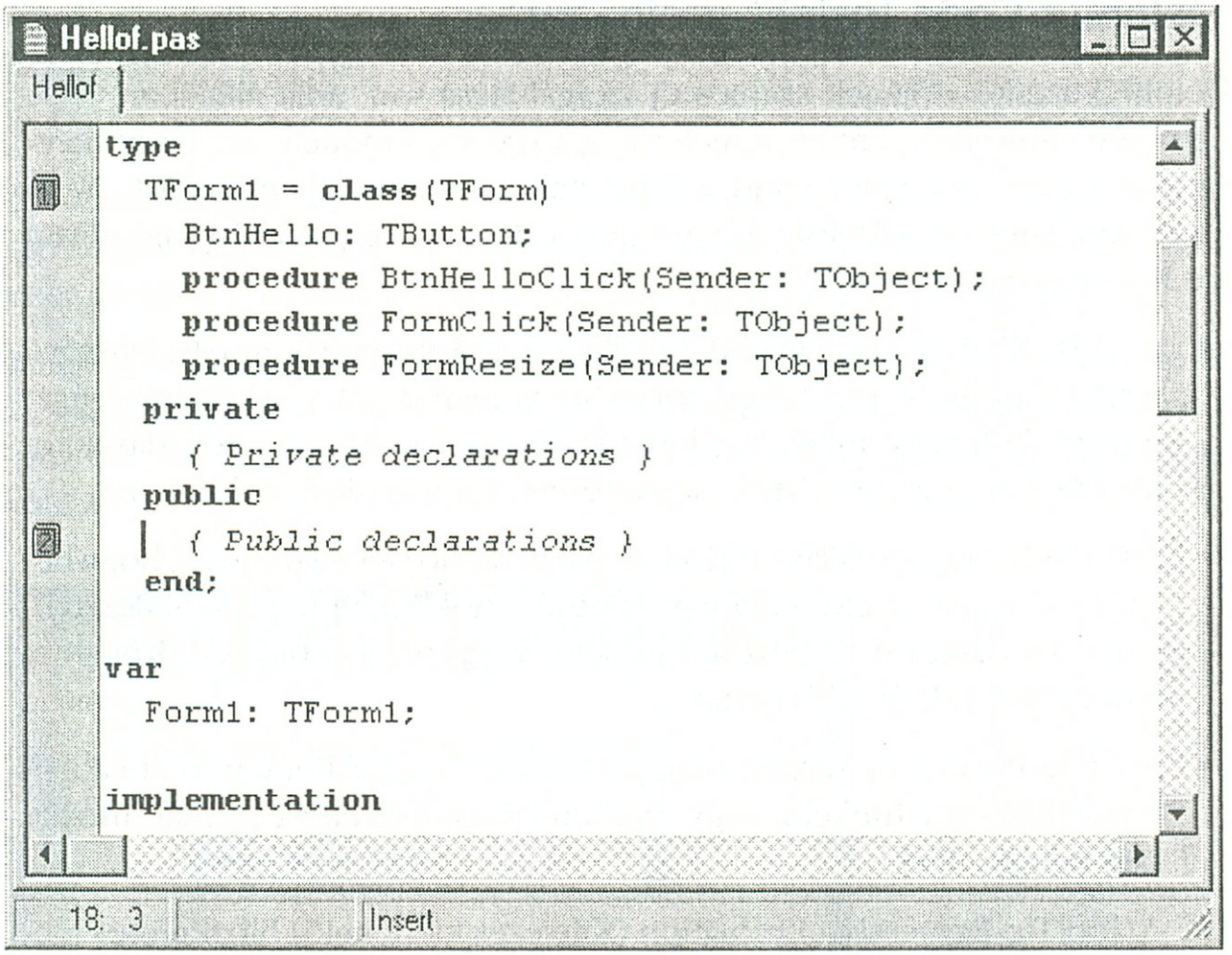
A Delphi szerkesztőjében egyszerre több forráskód fájlon is dolgozhatunk, amelyek között a szerkesztő alján található „fülekkel" válthatunk. A szerkesztő minden lapja egy külön fájlhoz tartozik. Dolgozhatunk formokhoz tartozó egységeken, független Pascal egységeken, projekt fájlokon, a formok szöveges változatán, vagy akár sima szövegfájlokon is. A szerkesztő lapjai között a CTRL+TAB billentyűkkel válthatunk (a SHIFT+CTRL+TAB billentyűkkel pedig visszafelé). Amikor a szerkesztővel dolgozunk, akkor általában érdemes az ablakát úgy megnagyobbítani, hogy annyi sor látszódjon, amennyi csak lehetséges, és hogy csak ez, és az Object Inspector ablaka látszódjon, amikor programot írunk. Az Object Inspectorban azonnal láthatjuk a komponensek szerkesztési időben állítható tulajdonságainak nevét.

Van néhány környezeti beállítás, amelyek a szerkesztőre vonatkoznak, ezek az Environment Options párbeszédablak **Editor Options**, **Editor Display** és **Editor Colors** oldalain találhatóak többnyire. A **Preferences** oldalon a szerkesztő Autosave (automata mentés) szolgáltatását állíthatjuk be. A forráskód fájlok elmentése minden futtatás előtt megmentheti egy napi munkánkat, ha a programunk netán lefagyasztaná a rendszert (ami nem is olyan ritkán fordul elő, mint gondolnánk). A másik három oldalon megadhatjuk az alapvető szerkesztési opciókat, mint a billentyűk, a szintaxis kiemelési szolgáltatások, és a betűtípus beállítása. Ezeket a pontokat eléggé könnyű megérteni.

A szerkesztő ablak gyorsmenüjének van néhány parancsa a debuggoláshoz, és más parancsok, amelyek magára a szerkesztő ablakra vonatkoznak, olyanok, mint az aktuális lap bezárása, fájl, vagy egység megnyitása a kurzornál, az üzenet ablak megnézése és bezárása, valamint a szerkesztő opcióinak előhívása, amiről az előbb beszéltünk.

A szerkesztő könyvjelzőinek használata

A Delphi 3 szerkesztőjében soronként beállíthatunk könyvjelzőket. Amikor a szerkesztő egy során állunk, a CTRL+SHIFT és egy számbillentyűt megnyomva 0-9-ig egy könyvjelzőt definiálhatunk, amit a sor elején jelöl is (2.10. ábra) a rendszer. A megadott könyvjelzőre a CTRL és a számbillentyű megnyomásával ugorhatunk vissza. A CTRL+SHIFT a számbillentyű újbóli megnyomásával változtathatjuk a könyvjelző állapotát, azaz így vissza is vonhatjuk.



2.10. ábra *A könyvjelzők a Delphi szerkesztőjének bal oldalán lévő sávban*

A könyvjelzők meglehetősen hasznosak, ha egy hosszú fájlban több metódust szerkesztünk egyszerre, vagy az osztály definíciójáról az osztály metódusának a definíciójára akarunk ugrani.

A Borland cég nem dokumentálta a könyvjelzőket túl jól, valószínűleg azért, mert valami hibájuk lehet. Ha újra beállítunk egy már megadott könyvjelzőt, akkor azt odamozgatja az új helyre. Ez érthetőnek tűnik, de valójában ez probléma: ha egy új könyvjelzőt hozunk létre, és véletlenül egy már meglévő könyvjelző számát használtuk, akkor az előző pozícióról elvész a könyvjelző. Egy másik furcsa viselkedése a könyvjelzőknek, hogy egy sorra akár több könyvjelzőt is definiálhatunk, de csak az egyik jelét látjuk a sor elején. Az igazi probléma azonban az, hogy a könyvjelzők pozíciói nem mentődnek el a fájllal együtt, és ha újra megnyitjuk a fájlt, azok nem lesznek benne, ezért ezeket csak egyszeri szerkesztés alkalmával használhatjuk.

A Delphi 3 Code Insight szolgáltatásai

A Delphi 3 szerkesztőjének számos új szolgáltatása van, amit általában Code Insight-ként ismernek. Ennek a technológiának az alapötlete az, hogy megkönnyítsék mind a kezdő, mind a tapasztalt programozók munkáját. Négy funkció van (mind új a Delphi 3-ban) amit a Borland cég összefoglaló néven Code Insight-nak hív:

- A Code Completion Wizard (Kódkiegészítő varázsló), amely lehetővé teszi, hogy kiválasszuk egy komponens metódusát vagy tulajdonságát egy listából, vagy annak kezdőbetűit begépelve. Ez egy értékadási kifejezésben a megfelelő érték megkeresésében is segítségünkre lehet.
- A Code Templates Wizard (Kódsablon varázsló) lehetővé teszi, hogy beszúrjunk egyet az előre definiált kódsablonokból a programunkba, például egy összetett kifejezést egy belső `begin-end` blokkal. Könnyen létrehozhatunk új sablonokat is.
- A Code Parameter Wizard (Kódparaméter varázsló) egy tipp-ablakban megjeleníti egy funkció, vagy metódus paramétereinek típusát, miközben beírjuk őket.
- A ToolTip Expression Evaluation (Kifejezés kiértékelő) hibakeresés ideje alatt használható szolgáltatás. Megmutatja egy azonosító, tulajdonság, vagy kifejezés értékét, amin az egér éppen áll.

A ToolTip Expression Evaluation szolgáltatásról később lesz szó a fejezetben, a hibakeresés bemutatása során. A következő három szakaszban a többi Code Insight funkciót beszéljük meg részletesebben. Az Environment Options párbeszédablak **Code Insight** oldalán engedélyezhetjük és letilthatjuk (vagy beállíthatjuk) ezeket a varázslókat.

Kódkiegészítés

Két módja van, hogy aktívájuk ezt a varázslót. Egyszerűen begépelhetjük egy objektum nevét, mint például Button1, és a végére egy pontot és várunk:

```
Button1.
```

A Delphi kiírja az objektum lehetséges tulajdonságainak és metódusainak listáját, ahogy azt a 2.11. ábra is mutatja. Az, hogy mennyit kell várnunk amíg a lista megjelenik, a **Code Completion** opción múlik, amit a **Code Insight** oldalon állíthatunk be.

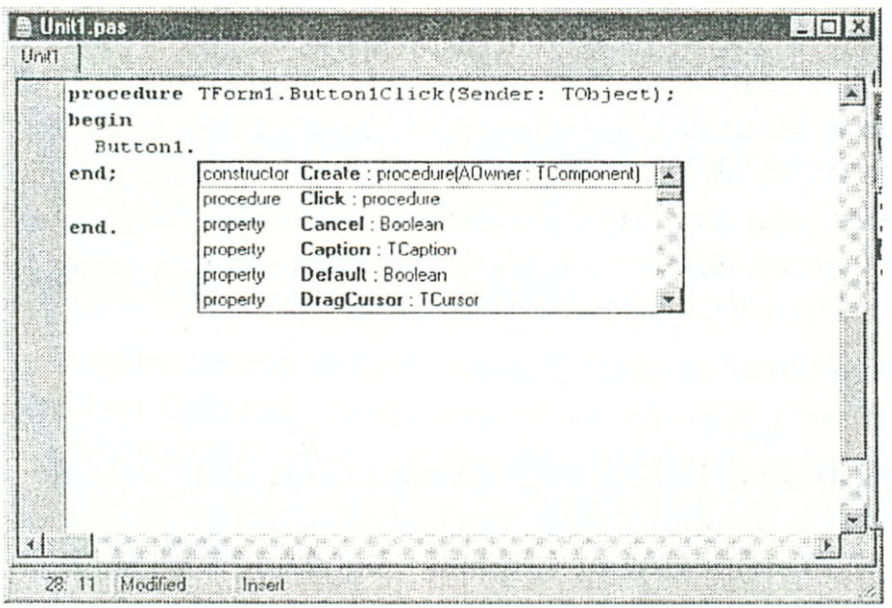
Alternatívaként beírhatjuk a tulajdonság vagy metódus kezdő betűit, mint Button1.Ca, és utána a CTRL+SPACE billentyűket, mire rögtön megjelenik a lista, de ez alkalommal a varázsló megpróbál rájönni, hogy melyik tulajdonságot,

vagy metódust keressük, a kezdőbetűk alapján. Ezt a billentyűkombinációt értékadási kifejezésben is használhatjuk. Ha beírjuk:

```
x :=
```

és utána megnyomjuk a CTRL+SPACE billentyűket, akkor a Delphi kiírja a lehetséges objektumok, változók, vagy konstansok listáját, amit a program ezen részében használhatunk (ami a jelenlegi hatáskörön belül van).

Megjegyzendő, hogy a Delphi dinamikusan dönti el, hogy milyen elemek látszódjanak a listában, mert a háttérben folyamatosan elemzi a programot, amit írunk. Ha egy új változót adunk a unithoz, akkor az meg fog jelenni a listában.



2.11. ábra *A Code Completion varázsló működés közben*

Kódsablonok

A kódkiegészítő varázslót, ellentétben a kódsablon varázslóval, manuálisan kell aktiválni. A CTRL+J billentyűk lenyomásával jeleníthetjük meg az összes sablont tartalmazó listát.

Leggyakrabban először beírunk egy kulcsszót, mint az `if` vagy `array`, és azután nyomjuk meg a CTRL+J billentyűket, hogy csak a kulcsszavakkal kezdődő sablonok jelenjenek meg a listában. Néhány kulcsszóhoz több sablont is készített a Borland cég, amelyek mind a kulcsszavakkal kezdődnek (pl.: ifA és ifB). Ha a kulcsszó után nyomjuk meg a CTRL+J billentyűket, akkor az összes sablon, ami a kulcsszóhoz kapcsolódik, meg fog jelenni.

A kódsablon varázslót használhatjuk arra is, hogy egy gyakori kifejezésnek nevet adjunk. Például ha a MessageDlg funkciót gyakran használjuk, akkor létrehozhatunk egy új sablont „mess" néven, egy rövid leírás, és a következő szöveg hozzáadásával:

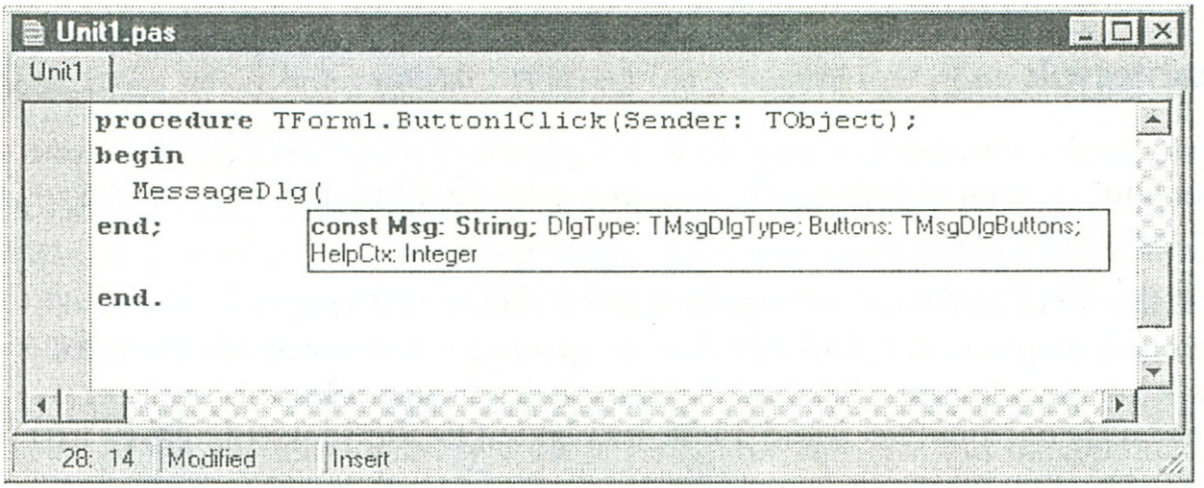
```
MessageDlg ('|', mtInformation, [mbOK], 0);
```

Mostantól, ha egy üzenetablakot akarunk létrehozni, csak be kell írni, hogy „mess", utána megnyomni a CTRL+J billentyűket, és megkapjuk az egész szöveget. A függőleges vonal (vagy más néven pipe, szűrő) karakter jelzi, hogy a programkód bemásolása után a Delphi hova helyezze a kurzort. Érdemes választani egy pozíciót, ahonnan folytathatjuk a sablon által generált program szerkesztését.

Ahogy ez a példa is bemutatja, a kódsablonoknak nincs közvetlen kapcsolatuk a nyelvi kulcsszavakkal, annál általánosabb műveletek. A kódsablonok a DELPHI32.DCI fájlban vannak elmentve, így lehetséges, hogy a kódsablonokat ebben a fájlban más gépekre is átmásoljuk. Nincs mód két kódsablon fájl összefűzésére, és nincs külső cég által készített eszköz sem még, hogy több sablont adjunk hozzá egy géphez. Ezekre a bővítésekre várnunk kell, amíg a Borland cég nem dokumentálja a DCI fájl belső felépítését.

Kódparaméter

A harmadik, és talán a legjobb Code Insight technológia a Code Parameter. Régebben, amikor egy ismeretlen eljárást kellett meghívnunk, akkor az eljárás nevének begépelése után az F1-et megnyomva kellett megnéznünk a Helpben a paraméterezését. Most egyszerűen beírjuk annak nevét és az azt követő nyitó zárójelet, mire a paraméterek nevei és típusai rögtön megjelennek a tipp-ablakban, ahogy azt a 2.12. ábrán láthatjuk.



2.12. ábra *Az új Code Paramater varázsló kiírja a funkciók és eljárások paramétereinek típusát, miközben beírjuk őket*

Az ábrán félkövéren van az első paraméter kiírva. Az első paraméter, és az azt követő vessző beírása után a második paraméter lesz félkövéren kiírva, és minden egyes paraméter beírását követően így tovább. Ez a sok paraméterrel rendelkező eljárásoknál nagyon hasznos, mint például a Windows API néhány függvényénél. Próbáljuk ki a `CreateWindow(` beírásával, és megértjük, hogy miért.

A Code Parameter varázsló is a háttérben elemzi az általunk beírt programot. A következő eljárást írjuk egy egység implementációs részének elejére:

```
implementation

procedure ShowInt (X: Integer);
begin
  MessageDlg (IntToStr (X),
    mtInformation, [mbOK], 0);
end;
```

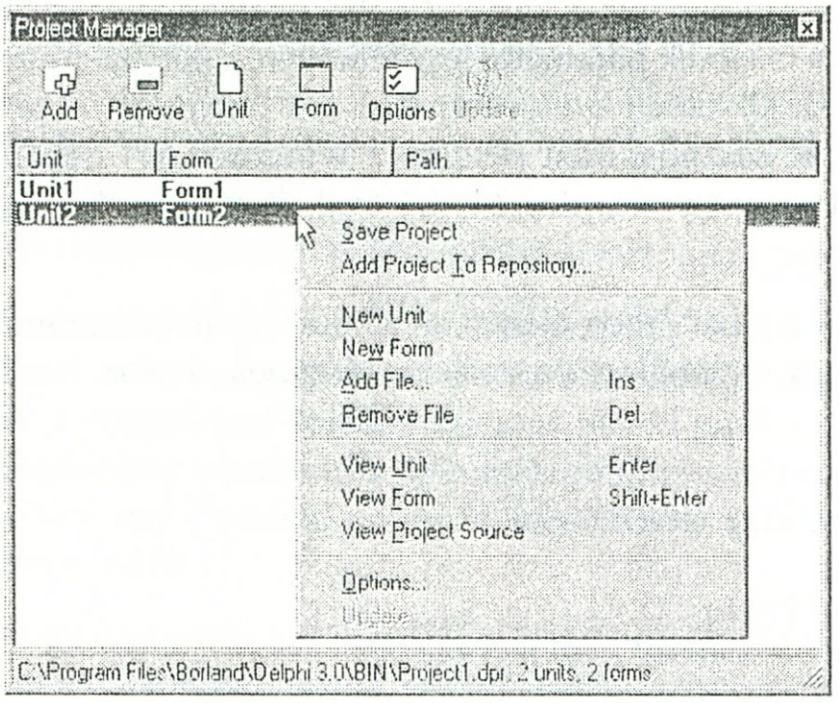
Ezután meglesz az összes szükséges információnk az eljárás paramétereiről, ha a későbbiekben beírjuk, hogy `ShowInt(`.

Projektek menedzselése

A Delphit használva tudnunk kell, hogyan kezeljük a projekt fájlokat. Az 1. fejezetben láthattuk, hogy megnyithatunk egy projekt fájlt a szerkesztőben, és szerkeszthetjük. Van egyszerűbb módja is a projekt néhány beállításának megváltoztatására, például a Project Manager ablakban a **Project Options** használatával.

A Project Manager

Amikor egy projektet betöltünk, akkor a **View** menü **Project Manager** parancsával egy projekt ablakot nyithatunk meg. Az ablakban az aktuális projektet alkotó összes form és egység listáját láthatjuk. A 2.13. ábrán egy ilyen ablak példáját láthatjuk, amelyben két különböző formból álló projekt listája látható. A Project Manager gyorsmenüjével a projekten végezhetünk néhány műveletet, mint például új vagy meglévő fájlokat adhatunk hozzá, fájlokat törölhetünk, megnézhetünk forráskód fájlt, vagy formot, és a projektet hozzáadhatjuk a Repository-hoz. A legtöbb ezek közül megtalálható az ablak SpeedBar-ján is.



2.13. ábra *A Project Manager ablak a saját gyorsmenüjével*

Projektopciók beállítása

A Project Manager-ből (vagy a **Project** menüből) előhívhatjuk a Project Options párbeszédablakot. A Project Options első, **Forms** nevű lapja listázza a program indulásakor automatikusan létrehozandó, és a program által létrehozott formokat. Könnyedén átmozgathatjuk a formokat az egyik listából a másikba. Az **Application** oldalon állíthatjuk be az alkalmazás nevét, ikonját, és súgófájljának nevét. A többi beállítás a Delphi fordítójához és szerkesztőjéhez, verzió információkhoz, és a futásidejű csomagok használatához kapcsolódik.

Megjegyzés

Két mód van a fordító opcióinak beállításához. Az egyik a Project Options **Compiler** oldalának használata, a másik a direktívák forráskódban való beállítása `{$X+}` vagy `{$X-}` formában (X az adott direktíva betűjele). A második megközelítés sokkal rugalmasabb, mivel így lehetséges, hogy egy opciót csak adott forráskód fájlokon állítsunk be, vagy akár csak egy pár programsorra.

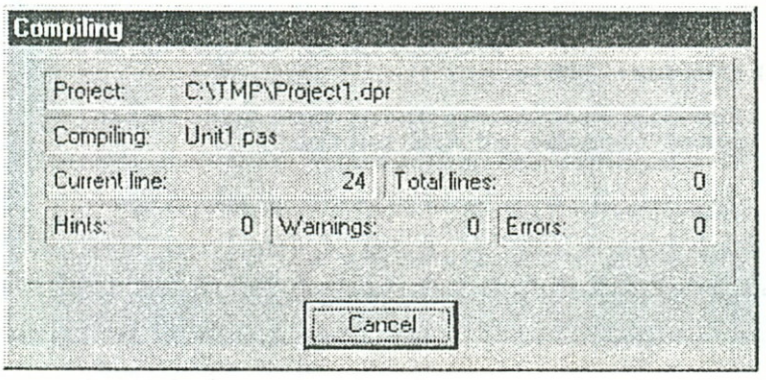
Az összes beállítás automatikusan mentésre kerül a projekttel, egy külön DOF kiterjesztésű fájlba. Ez egy sima szövegfájl, amit könnyedén szerkeszthetünk. Nem szabad letörölni ezt a fájlt, ha bármelyik opció alapértelmezés szerinti értékét megváltoztattuk.

Egy projekt lefordítása

Több módon fordíthatunk le egy projektet. Ha futtatjuk (az F9-et megnyomva, vagy a SpeedBaron lévő ikonra kattintva) a Delphi először lefordítja, később csak az időközben megváltozott fájlokat fordítja újra a rendszer. Ha viszont a **Compile** menü **Build All** parancsát választjuk, akkor az összes fájl fordításra kerül, akkor is, ha nem változtattunk rajtuk. Ezt a parancsot ritkán használjuk, mert a Delphi képes eldönteni, hogy mely fájlok változtak meg, és szükségszerűen lefordítja őket. Az egyetlen kivétel, ha megváltoztatunk valamit a projekt opciókban. Ebben az esetben a **Build All** parancsot kell használnunk, hogy az új opciók érvényesüljenek.

A projekt tartalmazza a hozzátartozó forráskód fájlokat és formokat. Ez a lista látható a Project Source és Project Manager ablakokban, és arra használható, hogy lefordítsunk, vagy újra építsünk egy projektet. Először minden forráskód fájl egy Delphi egységgé (Delphi Compiled Unit) alakul, a Pascal forráskód fájl nevével, és DCU kiterjesztéssel. Például a UNIT1.PAS lefordított változata a UNIT1.DCU.

Amikor a projekt saját forráskódja is le van fordítva, akkor a projektet alkotó lefordított egységek, és a VCL könyvtár részei egy futtatható fájllá szerkesztődnek (linkelődnek) össze. A fordítási lépéseket jobban megérthetjük, ha a **Show Compiler Progress** opciót engedélyezzük, mert így követhetjük, hogy mi történik a fordítás során. Ezt az opciót az Environment Options párbeszéd ablakának **Preferences** oldalán találhatjuk a **Compiling** fejléc alatt (2.9. ábra). Ámbár ez kissé lelassítja a fordítást, a Compile ablakban láthatjuk, hogy éppen melyik forráskód fájlt fordítja a rendszer (hacsak nem túl gyors a számítógépünk; a Delphi több fájlt is lefordíthat másodpercenként egy gyors PC-n). A 2.14. ábrán látható egy példa erre.



2.14. ábra *A fordítási folyamat ablaka*

Egy lefordított program vizsgálata

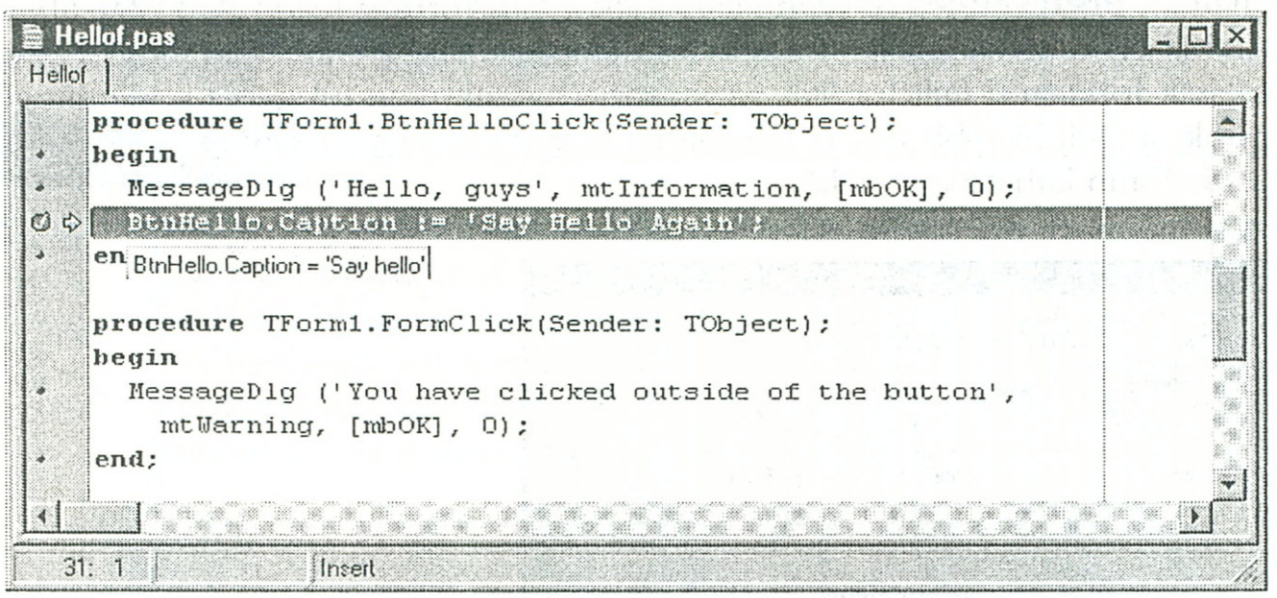
A Delphiben van néhány eszköz, amellyel egy lefordított programot vizsgálhatunk, ide értve a hibakeresőt és a Browsert. A következő részek röviden bemutatják ezeket az eszközöket, amelyeket a 26. fejezetben részletesen is tárgyalunk.

A beépített hibakereső

A Delphi beépített debuggerének (hibakereső) sok funkciója van, habár a Borland cégnek van egy sokkal jobb, különálló eszköze is, a Turbo Debugger. Majdnem az összes hibakeresési feladatunkat el tudja látni a beépített debugger, különösen ha a CPU View ablakot (26. fejezet) is használjuk. A különálló Turbo Debugger néhány speciális esetben lehet hasznos.

Nem kell sokat tennünk, hogy használhassuk a beépített debuggert. Valójában minden alkalommal, amikor Delphiben futtatunk egy programot, az ezzel az eszközzel indul el alapértelmezés szerűen. Ez azt jelenti, hogy a forráskód bizonyos pontján beállíthatunk egy töréspontot (breakpoint), melyet ha elérünk program futása során, akkor az ott megszakad. Például nyissuk meg a Hello2 programot, amit az 1. fejezetben készítettünk, és kattintsunk duplán a formon található gombon, hogy a megfelelő programrészre ugorjunk a forráskódban. Most beállíthatunk egy töréspontot, ha a szerkesztőben a margó melletti sávban kattintunk, a gyorsmenü **Toggle Breakpoint** parancsát választjuk ki, esetleg lenyomjuk az F5 billentyűt megnyomva.

A szerkesztőben a sornak, ahova a töréspontot tettük, megváltozik a színe. Most már futtathatjuk a programot, de minden alkalommal, amikor megnyomjuk a gombot, a debugger megállítja a programot, és a megfelelő programsort



2.15. ábra *A ToolTip Expression Evaluation a legegyszerűbb módja az objektumok értékeinek és tulajdonságainak vizsgálatára*

jeleníti meg. Innentől kezdve soronként (lépésenként) futtathatjuk a programot, láthatjuk a meghívott funkciók kódját, vagy folytathatjuk a program futtatását.

Amikor egy programot megállítunk, megvizsgálhatjuk részletesen az aktuális állapotát. Több mód is van egy érték vizsgálatára. A legegyszerűbb megközelítés a ToolTip Expression Evaluation. Egyszerűen csak mozgassuk az egeret bármelyik változó neve fölé, és láthatjuk éppen aktuális értékét egy kis tipp ablakban, ahogy azt a 2.15. ábra is mutatja (ahol egy töréspontot is láthatunk a szerkesztőben).

Tipp

Néha úgy tűnhet, hogy a ToolTip Expressions Evaluation nem működik. Ez akkor fordulhat elő, ha a fordító az optimalizálás során néhány változót a generált kódban a CPU regisztereibe tett. Ha a fordító optimalizálását kikapcsoljuk, akkor több ToolTipet kapunk.

Az Object Browser

Ha egyszer már lefordítottunk egy programot, akkor használhatjuk az Object Browsert, hogy megvizsgáljuk (még akkor is ha nem futtatjuk vagy debuggoljuk). Ezzel az eszközzel megnézhetjük a program által definiált összes osztályt (vagy az egységek által definiált osztályokat, amit a program direkt vagy indirekt módon használ), az összes globális nevet, változó nevet, és így tovább. Az Object Browser minden egyes osztálynál megmutatja az összes – mind a lokális, örökölt, private és public – tulajdonság, metódus, változó listáját. A kiírt információk nem mondanak sokat, ha nem vagyunk jártasak a Delphi által használt Object Pascal nyelvben. A 26. fejezetben visszatérünk az Object Browserre, miután átvettük a megértéséhez szükséges információkat.

Egyéb Delphi eszközök

A Delphiben sok más programozást segítő eszköz is van. Ilyen például a Menu Designer, a menü struktúrák kialakítására használható vizuális eszköz. Van sokféle varázsló is, amelyekkel egy alkalmazás struktúráját generálhatjuk, vagy új formot hozhatunk létre. A többi eszköz különálló kis program, melyek windowsos alkalmazások fejlesztéséhez kapcsolódnak, mint például az Image Editor és a WinSight, amivel a Windows üzenetek áramlását követhetjük figyelemmel.

Van számos külső adatbázis kezelő eszköz, mint a ReportSmith, a Database Desktop, és a Database Explorer, amelyek közül néhány nem található meg a Delphi alacsonyabb szintű kiadásaiban. Egy programozó használhat külső eszközöket is, hogy a Delphi gyengéit kiegészítse. Például használhatunk egy erőforrás-szerkesztőt (mint a Borland Resource Workshop-ja), vagy egy jobb Help fájl készítőt. Ezen kívül használhatunk még sok más külső cég által szállított Delphi eszközt. Néhány ilyen program demóverzióját megtalálhatjuk a CD-n, de ezeken kívül még sok más is beszerezhető. Néhány külső program igazán jól kiegészítheti a Delphit, és így segít minket, hogy hatékonyan tudjunk dolgozni.

A rendszer fájljai

Ahogy azt láthattuk, a Delphi sokféle fájlt generál minden egyes projektnek, és tudnunk kell, melyek azok, és mi a nevük. Alapvetően két olyan elem van, amely befolyásolhatja, hogy a fájloknak mi lesz a neve: a projekt és a formok neve, és az előre definiált fájl kiterjesztések, amelyeket a Delphi használ. A 2.2. táblázatban láthatjuk a kiterjesztések listáját, amiket egy Delphi projekt könyvtárában megtalálhatunk. A megtaláljuk azt is, hogy mikor keletkeznek ezek a fájlok, és mi a jelentőségük a későbbi fordítások során.

2.2. táblázat

Delphi fájl kiterjesztések

Kiterjesztés	Fájltípus és leírása	Létrehozás ideje	Szükséges a fordításhoz?
BMP ICO CUR	Bitmap fájl, ikon fájl, kurzor fájl: általános Windows fájlok, amik képeket tárolnak.	Fejlesztés	Amikor egy ImageEditor tulajdonság értékeként használjuk, akkor nem, de néhány alkalmazás futás közben betölti őket
DCP	Delphi Component Package: Egy csomagba fordított Pascal fájl szimbólum információit tartalmazza fordított kód nélkül.	Fordítás	Csomagok használatakor. Ez a fájl nem része a projektnek, és a Delphi egy csomag fordításakor hozza létre. DPL fájlokkal együtt szállítjuk.
DCU	Delphi Compiled Unit: egy Pascal fájl lefordításának eredménye.	Fordítás	Csak ha a forráskód nem elérhető. A DCU fájlok a fordítást meggyorsítják.

2.2. táblázat

Delphi fájl kiterjesztések (folytatás)

Kiterjesztés	Fájltípus és leírása	Létrehozás ideje	Szükséges a fordításhoz?
DFM	Delphi Form File: egy bináris fájl, amely a form (vagy egy adatmodul) komponenseinek tulajdonságait tartalmazza.	Fejlesztés	Igen. Ahogy korábban is és ebben a fejezetben is említettük, átkonvertálhatjuk szöveges formátumúvá.
~DF (DFM)	Backup of Delphi Form File	Fejlesztés	Nem. Ez a fájl akkor jön létre, amikor egy egység újabb verzióját mentjük el amely a formmal kapcsolatos.
DLL	Dynamic Link Library: egy futtatható fájltípus.	Fordítás/Linkelés	Lásd EXE
DOF	Delphi Option File: egy szövegfájl a projekt aktuális beállításaival. Delphi 1-ben OPT volt a kiterjesztése.	Fejlesztés	Nem. Csak akkor ha speciális fordítási opciók vannak.
DPK	Delphi Package: egy csomag, ami a projekt forráskódját tartalmazza.	Fejlesztés	Igen.
DPL	Delphi Package Library: egy DLL, mely Delphi komponenseket tartalmaz, amelyeket a Delphi környezet használ a szerkesztés alatt, vagy az alkalmazás futása alatt.	Fordítás/Linkelés	Lásd EXE. Más Delphi fejlesztőknek is átadhatók.
DPR	Delphi Project File: ez a file gyakorlatilag Pascal forráskódot tartalmaz.	Fejlesztés	Igen.
~DP	Backup of Delphi Project (DPR)	Fejlesztés	Nem. Ez a fájl automatikusan jön létre, ha a projekt fájl egy újabb verzióját elmentjük.

2.2. táblázat

Delphi fájl kiterjesztések (folytatás)

Kiterjesztés	Fájltípus és leírása	Létrehozás ideje	Szükséges a fordításhoz?
DSK	Desktop file: a Delphi ablakainak pozícióját, a szerkesztőben megnyitott fájlok, és más desktop beállításokat tartalmaz.	Fejlesztés	Nem. Gyakorlatilag törölnünk kell, ha a projektet más könyvtárba másoljuk.
DSM	Delphi Symbol Module: Az összes szimbólum böngésző információt tartalmazza.	Fordítás	Nem. Ezt a fájlt az Object Browser használja, ha még nem fordítottuk le, vagy nem tudtuk újrafordítani a projektet (ha beállítottuk az opciót). Egyébként a memóriából veszi az információkat.
EXE	Executable file: A Windows alkalmazás, amelyet készítettünk.	Fordítás/Linkelés	Nem. Ezt a fájlt terjesztjük. Ebben van az összes lefordított egység, forráskód és form. Újrafordítással hozhatunk létre újabb verziót.
HTM/HTML	Hyper Text Markup Language: Az Internetes honlapok nyelve. A Web Deployment folyamat megismétlésével hozhatjuk létre újra.	Delphi ActiveX	Nem. Ez nem tartozik bele a projekt fordításába.
OCX	OLE Control eXtension: Egy DLL speciális verziója, amely ActiveXet vagy formokat tartalmaz.	Fordítás/Linkelés	Lásd: EXE.
PAS	Pascal fájl. Egy Pascal egység forráskódja, amely vagy formhoz tartozik, vagy önálló.	Fejlesztés	Igen.

2.2. táblázat

Delphi fájl kiterjesztések (folytatás)

Kiterjesztés	Fájltípus és leírása	Létrehozás ideje	Szükséges a fordításhoz?
~PA	Backup of Pascal file Egy Pascal fájl biztonsági másolata.	Fejlesztés	Nem. Automatikusan létrejön, ha a forráskód egy újabb verzióját elmentjük.
RES	Resource file: Egy bináris fájl, amely általában a projekt ikonját tartalmazza. Más, ilyen típusú fájlt is hozzáadhatunk a projekthez.	Fejlesztés	Igen. Egy alkalmazás fő RES fájlja a Project Options párbeszédablak Application oldalán lévő beállításoktól függően építhető újra.
STR	String Resource file: Ez egy RES fájl, ami a kódban deklarált resourcestringek számára generált string táblázatot tartalmazza (4. fejezet).	Fordítás	Nem. A Delphi automatikusan generálja amíg a programot fordítja.
TLB	Type Library. Ez a fájl vagy automatikusan, vagy az OLE Type Library szerkesztője által jön létre.	Fejlesztés	OLE programoknak szükséges.

Ezen fájlok legtöbbje, a forráskód és biztonsági mentés fájlokat, projekt fájlokat, az opció és desktop fájlokat beleértve nagyon rövidek. A lefordított egységek egy kicsit nagyobbak, és általában a futtatható fájl a legnagyobb egy projekt könyvtárában, hacsak nincsenek debug információkat tartalmazó fájlok, mint például a DSM kiterjesztésűek.

A DSM fájlok tartalmazzák a Object Browser által használt információkat, még akkor is, ha megváltoztattuk a forráskódot (de csak az első sikeres fordítás után). Ez különösen hasznos lehet, ha véletlenül hibázunk a forráskódban, mert az megakadályozza, hogy új verziót építsen a fordító. A DSM fájlok könnyen nagyon naggyá válhatnak, ezért az Environment Options párbeszédablak **Preferences** oldalán, a **Desktop Only** opciót válasszuk ki a **Desktop and Symbols** opció helyett, hogy ne generálódjanak DSM fájlok. Ez valamelyest meg is gyorsítja a fordítást.

Új!

Ahogy a 2.2. táblázatban is láthatjuk, a DPK fájlok forráskód csomagok, amíg a DPL és DCP fájlok egy csomag lefordításának eredményei. A Delphi 3 képes OCX fájlok és akár HTML fájlok generálására is, amely egy ActiveX kontrol vagy form kifejlesztéséhez szükséges. Más típusú fájlok, amelyeket a környezet hoz létre, az OLE típus könyvtárak (TBL) és a string forrás fájlok (STR).

A Delphi egyik előnye más vizuális programozói fejlesztő eszközökkel szemben az, hogy a forráskód fájlok egyszerű ASCII szövegfájlok. A Pascal forráskódot, projekt kódot és a formokat leíró fájlokat már átnéztük az 1. fejezetben. Most nézzük át az opció és desktop fájlok struktúráját. Mindkét fájltípus a Windows INI fájlokhoz hasonló felépítésű, amelyekben minden szakaszt egy szögletes zárójelek közt lévő név jelez. Például a Hello2 példaprogram HELLO.DOF fájljának egy részlete:

```
[Compiler]
A=1
B=0
...
[Linker]
MapFile=0
MinStackSize=16384
MaxStackSize=1048576
...
[Directories]
OutputDir=
SearchPath=
```

Megjegyzés

Delphi 2 és Delphi 3-ban az opció fájlok kiterjesztése a DOF, míg a Delphi 1-ben OPT. Ezeknek a fájloknak különböző a tartalma, és nem kompatibilisek.

Ennek a fájlnak az inicializáló része, amit kihagytam, fordítási opciók hosszú listája. A desktop fájlok által használt struktúra ugyanilyen, csak általában sokkal hosszabb. Érdemes megnézni, hogy mi van ezekben a fájlokban, hogy

jobban megértsük a szerepüket. Röviden egy desktop fájl (.DSK) Delphi ablakok pozícióját és állapotát listázza. Például ez a fő ablak leírása:

```
[MainWindow]
Create=1
Visible=1
State=0
Left=2
Top=0
Width=800
Height=97
```

Ez a néhány szakasz más ablakokra vonatkozik:

```
[ProjectManager]
[AlignmentPalette]
[PropertyInspector]
[Modules]
[formxxx]
[EditWindowxxx]
[Viewxxx]
```

A környezeti opciók és az ablakok pozíciói mellett a desktop fájl tartalmaz néhány fájlról (bizonyos típusúakról) adatokat visszamenőleg, a jelenlegi töréspontok helyét, aktív modulokat, bezárt modulokat és formokat.

Hogyan tovább?

Ebben a fejezetben a Delphi programozói környezetéről kaphattunk átfogó képet, tippeket és javaslatokat egyaránt. Egy új programozói környezethez hozzászokni eltart egy darabig, de az igazán érdekes és hasznos probléma az, hogy valójában hogyan írjunk programot.

A Delphi környezetét a Súgón keresztül jobban is megismerhetjük, mert ott találhatunk információkat a környezeti elemekről, ablakokról és parancsokról. Érdemes átnézni a súgófájlokban található leírásokat. Természetesen a legjobban akkor ismerhetjük meg a működését, ha programokat írunk. Most már továbbléphetünk a Delphi környezet egyik fontos szolgáltatására, amit eddig csak futólag említettünk, az Object Repository-ra és a varázslókra.

3. FEJEZET

Az Object Repository és a Delphi varázslók

E fejezet tartalma:

- Az Object Repository
- Létező alkalmazások és formok újrafelhasználása
- Az Adatbázis Form varázsló
- Egyéb Delphi varázslók
- Az Object Repository beállítása

Az Object Repository és a Delphi varázslók

Amikor új alkalmazás készítéséhez kezdünk (vagy csak egy új formhoz), két választásunk van. Elkezdhetjük elölről, egy üres alkalmazásból vagy formból, vagy választhatunk az Object Repository-ban (objektum tárló) lévő előre definiált modellekből. Ha a modellek mellett döntünk, újabb lehetőségeink vannak. Lemásolhatunk egy már létezőt (ezek voltak a sablonok a Delphi 1-ben), örököltethetünk egy meglevőből, vagy használhatjuk a varázslókat.

A varázsló egy programkészítő, régebben a Borland ezeket szakértőknek (expert) hívta, de a Delphi 3-ban a Microsoft hagyományoknak megfelelő varázsló nevet kapták. A varázslók feltesznek néhány kérdést, és a kapott válaszoknak megfelelően, néhány egyszerű szabály betartásával egyszerű programot készítenek. Ezután elkezdhetünk dolgozni a projekten, ami már tartalmaz némi kódot és komponenseket. Általában az így generált forrást azonnal le lehet fordítani, és ezek egy alapstruktúrát képeznek, amire ráépíthetjük programunkat vagy formunkat.

Ennek a rövid fejezetnek az a célja, hogy bemutassa az Object Repository-t, a Delphi varázslókat, és hogy megmutassa, milyen könnyű a használatuk. Nem mutatjuk be az általuk generált kódot, hiszen ezzel a könyvben még sok példa fog foglakozni. A programozó szemszögéből a varázslók nagyon hasznosak, probléma csak akkor van, ha állandóan ezeket használjuk, anélkül, hogy megértenénk, mit is tesznek. Emiatt sok példában a kódot kézzel készítjük el a megfelelő varázsló használata helyett.

Az Object Repository

A Delphiben több menüpont szolgál arra, hogy új formot, alkalmazást, adatmodult, komponenst, és ezekhez hasonló objektumokat hozzunk létre. Ezeket a parancsokat a **File** menü, és néhány más legördülő menü tartalmazza. Mi történik, ha a **File** menü **New** parancsát kiválasztjuk? A Delphi megnyitja az Object Repository-t (amit **New** párbeszédablaknak is hívnak). Az Object Repository segítségével akármilyen új elemet létrehozhatunk: formokat, alkalmazásokat, adatmodulokat, könyvtárakat, thread (szál) objektumokat, komponenseket, OLE Automation objektumokat, stb. Az Object Repository ablaknak több lapja van:

- A **New** és az **ActiveX** oldalakon sok különböző új dolgot hozhatunk létre. Amikor valami újat készítünk, a Delphi megkérdezi az új osztály nevét, és néhány egyéb dolgot, akár egy varázsló.
- Az „aktuális projekt" oldal (itt az aktuális projekt nevét láthatjuk) lehetővé teszi, hogy egy, az aktuális projektben szereplő formból vagy adatmodulból örököltessünk.

- A **Forms**, **Dialogs** és a **Data Modules** lapokon készíthetünk új elemeket ezekből a típusukból, egy már meglévőből kiindulva, vagy egy varázsló segítségével.
- A Projects oldalon állományokat másolhatunk ki a Repository-ban elhelyezett, már létező projektekből, vagy használhatjuk az Application varázslót.
- Az Object Repository alján lévő rádiógombok segítségével állíthatjuk be, hogy másolni kívánjuk az adott elemet, örököltetni akarunk belőle, vagy másolat készítése nélkül, közvetlenül fel akarjuk használni.

Megjegyzés

Az ötödik fejezet és az A függelék az objektumorientált programozás öröklődésének elméletébe vezet be, a tizenkettedik fejezet pedig a Delphi formok örököltetésének részleteivel foglalkozik. Röviden, ez egy módszer arra, hogy egy létező formot új képességekkel ruházzunk fel, anélkül, hogy teljes egészében lemásolnánk. Ezzel a módszerrel az eredeti formot érintő változtatások megjelennek a belőle származókban is.

Ha varázslót választunk sablon helyett, akkor az egyetlen elérhető rádiógomb a **Copy**, ami azt jelenti, hogy végül egy új másolathoz, a generált kódhoz jutunk. Azt, hogy egy Object Repository elem varázslót jelképez, az ikonján lévő villanykörtéről állapíthatjuk meg.

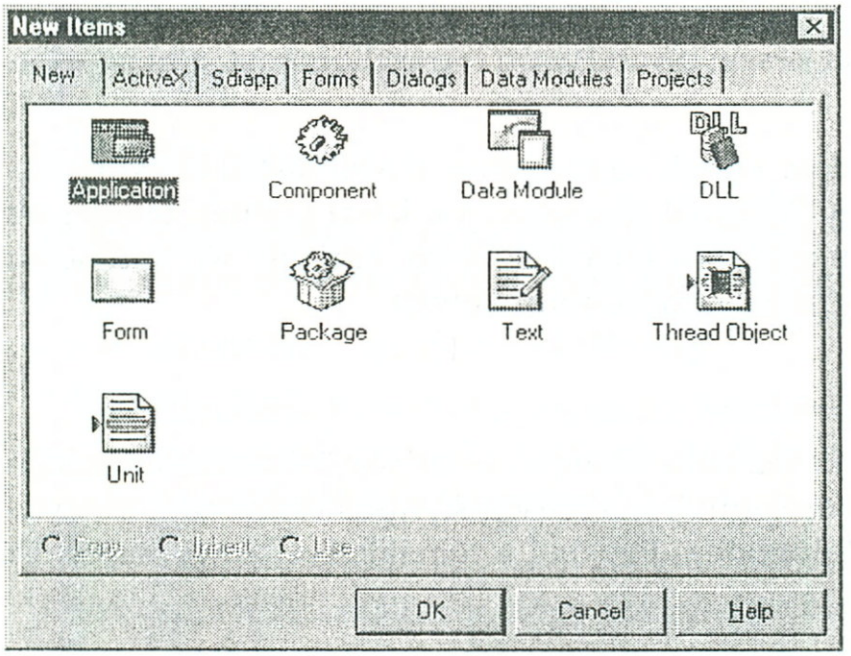
Ne felejtsük el, hogy itt a Delphi Client/Server változatának Object Repository lapjaira hivatkozunk, más változatokban kevesebb oldal és kevesebb elem szerepel, de a Repositoryt még tovább alakíthatjuk, amint azt ebben a fejezetben látni fogjuk.

Tipp

Az Object Repositorynak megvan a saját menüje, amivel az elemeket különböző szempontok (név, szerző, dátum, vagy leírás) alapján rendezhetjük, és különböző nézeteket (nagy ikonok, kis ikonok, lista, részletes) állíthatunk be. Az utolsó nézet az egyetlen, amiben láthatjuk a leírásokat, a szerző nevét, és az eszköz dátumát (ahogy az a 3.2 ábrán látható). Ez az információ különösen fontos olyan varázslók, projektek és formok esetében, amiket mi adtunk a Repositoryhoz.

A New oldal

Az Object Repository **New** oldala (3.1. ábra) segítségével sokféle, a gyakrabban használt típusokba tartozó új elemet hozhatunk létre, ennek tagjai gyakran menü parancsokat helyettesítenek. Itt következik egy lista azokról az elemekről, amiket ezen az oldalon készíthetünk:



3.1. ábra *Az Object Repository New oldala*

- *Application:* új, üres projektet hoz létre (ugyanaz, mint a **File** menü **New Application** pontja).
- *Component:* új Delphi komponenst készít, miután kitöltöttük a Component varázsló kérdéseit. Ugyanezt a varázslót hívhatjuk elő a **Component** menü **New Component** parancsával. A 18 fejezet ismerteti az új Delphi komponensek létrehozásának menetét.
- *Data Module:* új, üres adatmodult hoz létre (ugyanaz, mint a **File** menü **New Data Module** parancsa).
- *DLL:* egyszerű DLL vázat hoz létre, amint azt a 20. fejezetben láthatjuk.
- *Form:* új, üres formot készít (ugyanaz, mint a **File** menü **New Form** parancsa).
- *Package:* új Delphi komponenscsomagot készít (lomponens készítésekor is létrehozhatunk új csomagot). A csomagokról több fejezetben is szó lesz, de legtöbbet a 18. fejezetben, a komponens készítésének tárgyalásakor tudhatunk meg róla.

- *Report:* új, üres QuickReport (gyorsjelentés) formot készít. A jelentésekről és a nyomtatásról a 28. fejezetben lesz szó.
- *Text:* új ASCII szövegállományt nyit meg a Delphi szerkesztőben.
- *Thread Object:* új szál objektumot hoz létre, miután kitöltöttük a New Thread Object párbeszédablakot. A párhuzamos programozást a 25. fejezetben mutatjuk be.
- *Unit:* új, üres egységet készít; olyan Pascal forrásállományt, ami nem tartozik formhoz.
- *Web Server Application:* ennek segítségével ISAPI/NSAPI add-in DLL-eket, önálló CGI vagy Win-CGI állományokat készíthetünk. Mindegyik esetben a Delphi egy egyszerű projektet készít, ami egy Web modulon (egy különleges adatmodul) alapul form helyett.

Az ActiveX oldal

A Repository **ActiveX** oldala (ez nem szerepel a Delphi 3 Standard verziójában), az ActiveX kontrolok és formok készítésének, típus és kontrol könyvtárak karbantartásának kiindulópontja. Ezekkel a témákkal a könyv harmadik része foglalkozik.

- *ActiveX Form:* új ActiveFormot ad az aktuális projekthez. Az ActiveForm olyan Delphi form, amit ActiveX kontrolként felhasználhatunk, ahogy azt az első fejezetben láthattuk. Ez a parancs elindítja az ActiveX Control varázslót.
- *ActiveX Control:* elindítja a teljes ActiveX Control varázslót, ami lehetővé teszi, hogy ActiveX kontrolt építsünk egy meglévő, ablak alapú Delphi komponens köré.
- *ActiveX Library:* egy ActiveX telepítéshez használható DLL állományt készít (valójában csak egy pár sort ad a projekt állomány kódjához).
- *Automation Object:* új OLE Automation szerverobjektumot készít, az Automation Object varázsló kérdéseire adott válaszok alapján. Készít egy néhány soros kódot, és megnyitja a Type Library Editort.
- *Property Page:* új tulajdonságlapot készít. A tulajdonságlap az ActiveX kontrolok által használt tulajdonságszerkesztő.
- *Remote Data Module:* új, a szerveren elhelyezkedő adatmodul fejlesztése. Ez a távoli adatmodul elérhető a kliens alkalmazások számára, amik ezen keresztül érik el az adatbázis adatait.
- *Type Library:* megnyitja a Type Library (Típuskönyvtár) Editort. A típuskönyvtárak olyan interfészek, amik segítségével az OLE szerverrel a kliensek gyors és biztos kapcsolatot tarthatnak fenn.

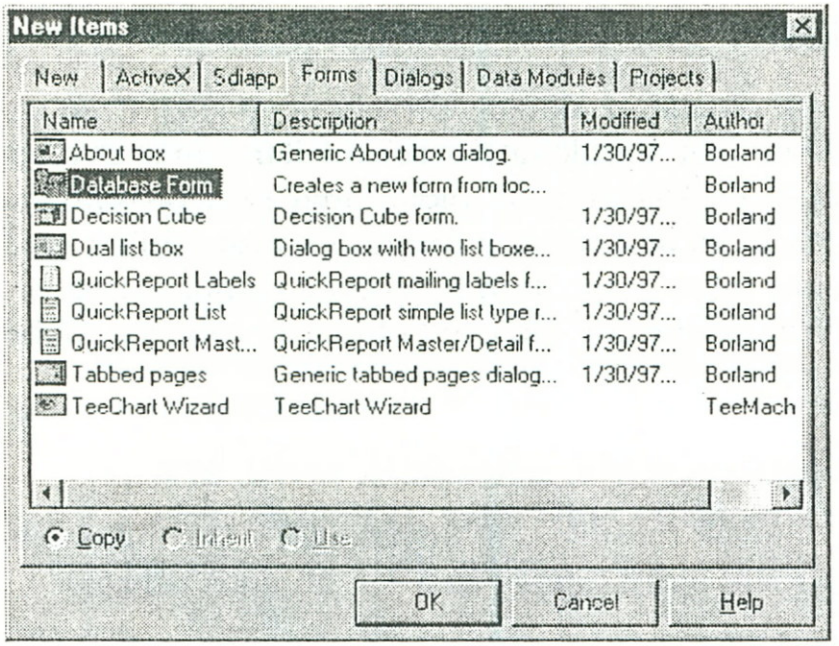
Az „aktuális projekt" oldal

Az Object Repository harmadik oldalát az aktuális projekt neve jelzi. Ezen az oldalon olyan új formokat, illetve adatmodulokat hozhatunk létre, amik egy, az aktuális projektben szereplőből öröklődnek. Az oldal tartalma a projektben szereplő egységektől függ. Ha készítünk néhány formot és visszatérünk ide, láthatjuk, hogy az oldal tartalma máris megváltozott. Viszont, amint bármit teszünk, az Object Repository azonnal bezáródik, ezért ehhez a próbához többször is meg kell azt nyitni.

A Forms oldal

Ez az oldal mutatja az előre definiált formokat. Itt következik a Delphiben elérhető előre definiált formok rövid listája (ahogy az a 3.2. ábrán is látható):

- *About box:* egy egyszerű Névjegy.
- *Database form:* bekapcsolja az Adatbázis Form varázslót, ez az egyik legfontosabb Delphi varázsló. Erről a varázslóról még részletesebben lesz szó ebben a fejezetben.
- *Dual list box:* ez egy form két különböző listadobozzal, ami lehetővé teszi a felhasználónak, hogy az egyikben elemeket válasszon ki, és átmozgassa azokat a másikba egy gombnyomással. A komponensek mellett ez a form elég sok nem triviális Pascal kódot tartalmaz.
- *Decision Cube:* egy, az új Decision Cube komponensen alapuló, formot hoz létre. Ez csak a Delphi Client/Server verziójában van meg, és ebben a könyvben nem foglalkozunk vele.
- *QuickReport Labels:* QuickReport komponensen alapuló jelentés formot készít. A jelentésekről a 28. fejezetben lesz szó.
- *QuickReport List:* egy másik, a QuickReport komponensre alapuló jelentés formot készít másfajta elrendezéssel.
- *QuickReport Master/Detail:* a harmadik előre definiált jelentés form, összetettebb struktúrával.
- *Tabbed pages:* a Windows 95 PageControl-jára (több oldalas kontrol) alapuló form. Erről a komponensről részletesebben a 14. fejezetben beszélünk.
- *TeeChart Wizard:* jó kiindulópont TeeChart grafikus komponenseken alapuló formok fejlesztésénél. Ez a komponens új a Delphi 3-ban, és a 11. fejezetben fogjuk tárgyalni.



3.2. ábra *A Repository Forms oldalán előre elkészített formokat és formvarázslókat láthatunk, részletes nézetben*

A varázslókat csak elindítani lehet, a többi form azonban többféle módon használható. Hozzáadhatóak projektekhez, örököltethetünk belőlük (ebben az esetben az eredeti form automatikusan a projektünkhöz adódik), vagy közvetlenül is felhasználhatók. Ha formot használunk, vagy örököltetünk egyet, figyeljünk arra, hogy soha ne változtassuk meg a tárlóbeli eredetit.

A Dialogs oldal

Ez az oldal hasonlít az előzőre, de olyan formokat tartalmaz, amiket általában párbeszédablakokhoz használunk. Ezek a következők:

- *Dialog with help:* két változata van. Az egyiknél a gombok a form jobb oldalán vannak, a másiknál pedig az alsó részen, ahogy az a megfelelő ikonon is látszik.
- *Dialog Wizard:* egy egyszerű varázsló, ami egy vagy több oldalas párbeszédablakok készítésére alkalmas, ahogy azt ebben a fejezetben látni fogjuk.
- *Password dialog:* egyszerű párbeszédablak, egy beviteli mezővel, amiben jelszót kérhetünk be.
- *Standard dialog:* szintén két változata van, különböző helyen lévő gombokkal.

A párbeszédablakok valójában formok, csak van néhány speciális tulajdonságuk (mint például a különleges keret), ahogy azt a 13. fejezetben látni fogjuk.

A Data Modules oldal

Azt már tudjuk, hogy mi a projekt, és mi a form, de mi az adatmodul? Ez egy olyan form, ami futásidőben soha nem jelenik meg a képernyőn, és nem vizuális komponensek tárolására lehet használni. Általában adatbázis-eléréshez szükséges kód fejlesztéséhez használjuk, ahogy az a 18. fejezetben látható lesz.

Ezen az oldalon kezdetben csak egyetlen modul van, a Customer Data. Ha több olyan alkalmazásunk, vagy formunk van, amik ugyanazokat a táblákat vagy lekérdezéseket használják, akkor könnyen készíthetünk új adatmodulokat és hozzáadhatjuk azokat a tárlóhoz.

A Projects oldal

Ez az utolsó oldal tartalmazza azokat a projekt sémákat, amiket saját alkalmazásaink kiindulópontjaként alkalmazhatunk, ezek gyakran egy vagy több formot tartalmaznak. Ezek a következők:

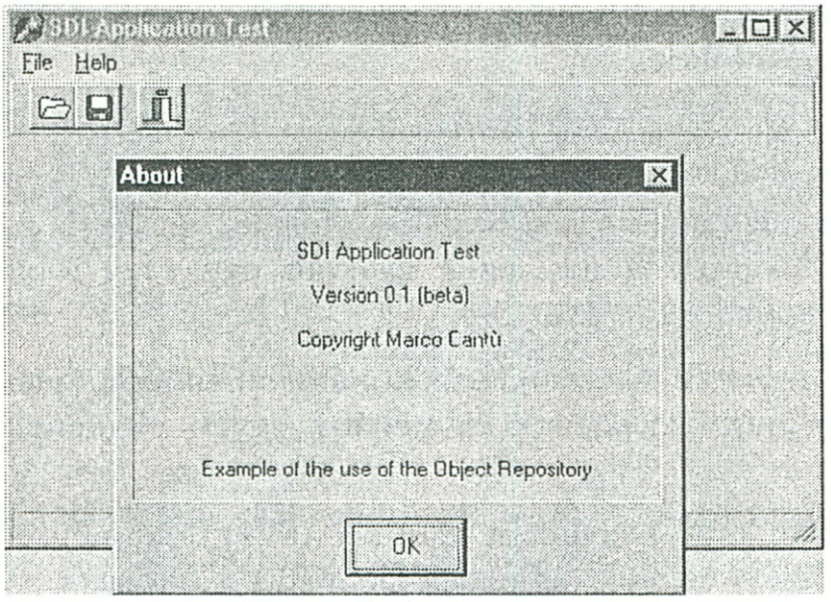
- *Application Wizard:* egyszerű varázsló, ami korlátozott módon segít egy alkalmazás állománystruktúrájának és néhány egyéb elemének megválasztásában.
- *MDI Application:* egy Multiple Document Interface (több dokumentumot kezelő) program főbb elemeit definiálja. Definiálja a főformot az MDI keretablakhoz, menüvel, státuszsorral, és eszköztárral. Létrehoz továbbá egy második formot, aminek segítségével futásidőben több gyermek-ablakot készíthetünk. Az MDI alkalmazásokat a 15. fejezetben fogjuk tanulmányozni.
- *SDI Application:* (egy dokumentumos alkalmazás) egy modern felhasználói felülettel rendelkező főformot definiál, státuszsorral, eszköztárral, és egy általános Névjeggyel.
- *Win95 Logo Application:* olyan példaalkalmazást készít, ami rendelkezik a legtöbb feltétellel ahhoz, hogy az alkalmazás megkaphassa a Windows 95 Logo-t. Ez a parancs egy SDI alkalmazást készít RichEdit komponenssel, és elkészíti azt a kódot, ami az alkalmazást felkészíti arra, hogy postázni tudjon.

Amikor ezek közül a projektek közül kiválasztunk egyet, a Delphi megkér, hogy írjuk be a nevét egy már létező vagy új mappának. Ha új mappát adunk meg, akkor a Delphi azt automatikusan létrehozza.

Például készíthetünk egy SDI projektet, ami a megfelelő sablonon alapul, ezután testre szabhatjuk, megfelelő címet adhatunk neki, hozzáadhatunk és elvehetünk belőle komponenseket, valamint írhatunk hozzá kódot. Sok érdekes komponens már alapból benne lesz, és ezekhez már van kész kód is, hogy működhessenek. Az alkalmazás menüje és eszköztára segítségével

párbeszédablakokat nyithatunk meg. A File Open és a File Save párbeszédablakok komponensekben vannak, ahogy azokat a Delphi a sablonokból a formhoz adta, a Névjegyet pedig egy második formon találhatjuk.

Az SdiTemp példában néhány egyszerű módosítást végeztünk: új címet adtunk a főformnak, és beírtunk néhány sor szöveget a Névjegy (About box) címsorába (mindkét változtatáshoz csak a Caption tulajdonságra volt szükség). Az eredmény a 3.3. ábrán látható, és a CD-n is megtalálható.



3.3. ábra *Az SdiTemp példa képe, amit az SDIproject sablonnal készítettünk*

Delphi varázslók

Már létező kód másolása és újrafelhasználása mellett a Delphi lehetővé teszi azt is, hogy új formot, alkalmazást, vagy másfajta forrásállományt készítsünk varázslókkal. A varázslók (Wizards) (vagy Experts) néhány opció beállítása után válaszainknak megfelelő kódot készítenek, teljesen automatikusan.

Az egyik legfontosabb, előre elkészített varázsló az Adatbázis Form varázsló (Database Form Wizard), amit a **Database** menü **Form Wizard** menüpontjával, vagy az Object Repository **Forms** oldalán lévő ikonnal aktiválhatunk. Ezen kívül van még néhány másik varázsló is a Delphiben, mint például az Alkalmazás varázsló (Application Wizard), vagy a Dialógus varázsló (Dialog Wizard), és az előző részben is több varázslót említettünk az Object Repository oldalainak felsorolásakor.

Vásárolhatunk még külső cégek által szállított, telepíthető varázslókat (néhány érdekesebb van a CD-n is.) vagy, akár saját varázslókat is készíthetünk, ezt a

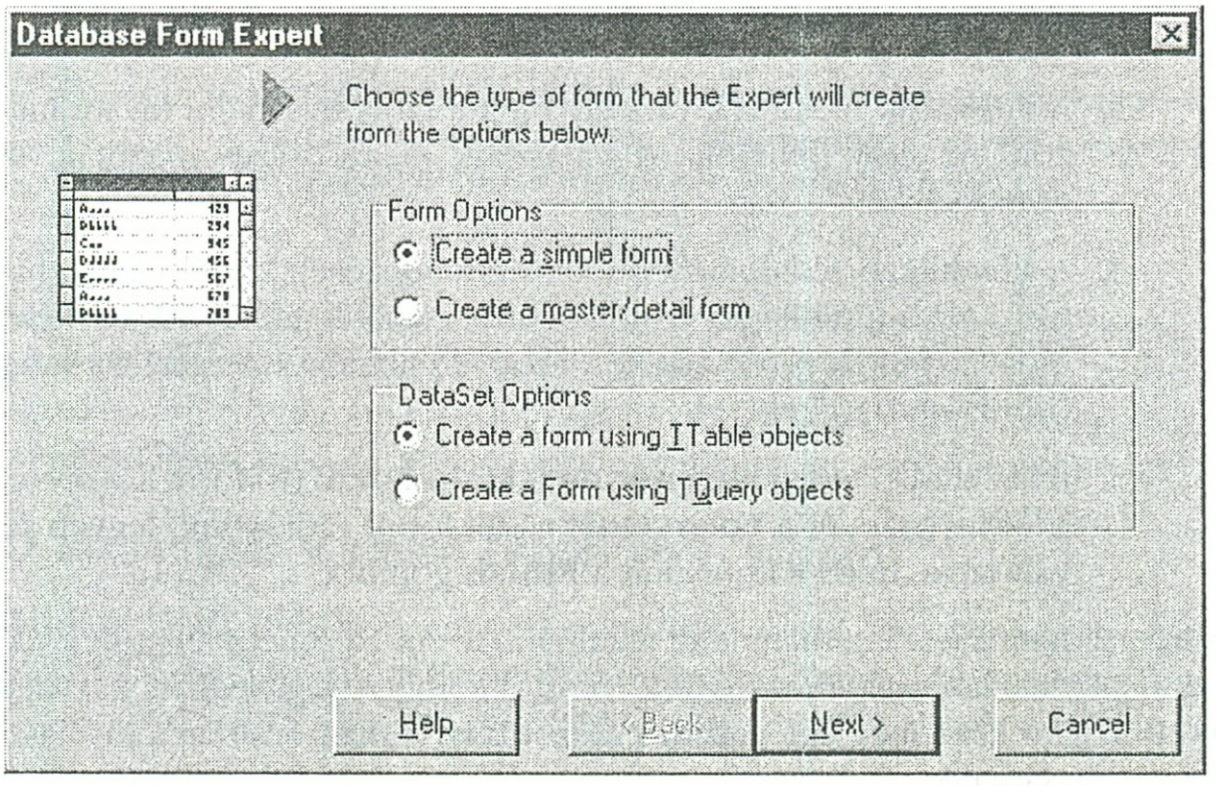
19. fejezetben tanulhatjuk meg. A telepíthető varázslók általában a **Help** menüben jelennek meg, de a Delphi többi legördülő menüjéhez is adhatunk új menüpontokat, vagy az Object Repository különböző lapjaira is telepíthetjük őket.

Az Adatbázis Form varázsló (Database Form Wizard)

Ebben a részben egy rövid példát láthatunk az Adatbázis Form varázsló használatára, de az elkészítendő alkalmazást nem írom le részletesen. Az adatbázisprogramozással kapcsolatban a 16. fejezet nyújt felvilágosítást.

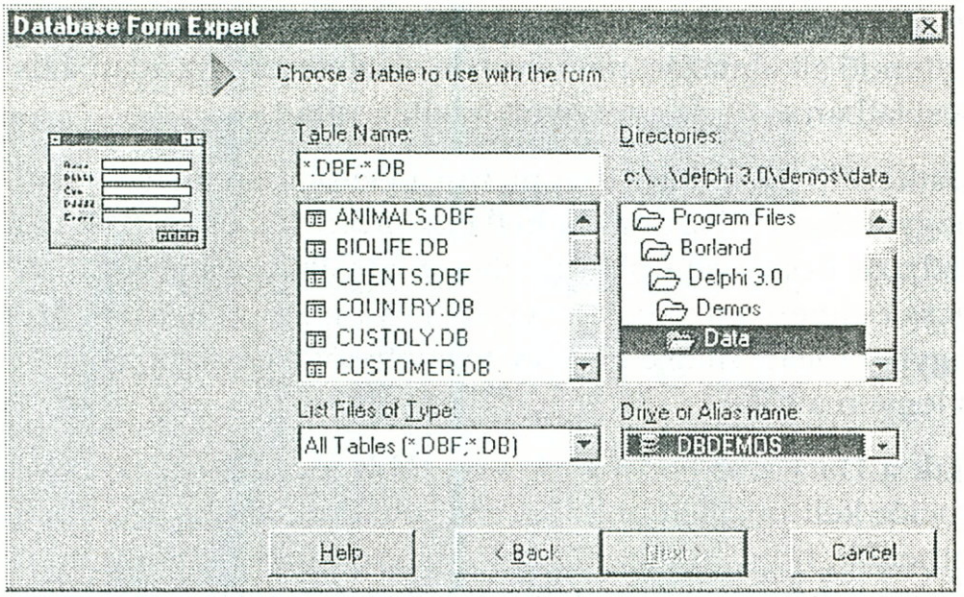
Ebben a példában adatkezelő programot fogunk írni a Delphiben megtalálható adatokkal. Fontos, hogy először készítenünk kell egy projektet, és csak azután indíthatjuk el az Adatbázis Form varázslót, úgyhogy általában mire végzünk, két formunk lesz, hacsak el nem távolítjuk az eredeti formot a projektből. Szerencsére a varázsló egyik utoljára megjelenő opciója lehetővé teszi, hogy az általa generált új form legyen a főform.

1. Amint elindítjuk a Database Form varázslót, több, egymással szorosan összefüggő opciót kell beállítanunk. Az első oldalon, ami a 3.4. ábrán látható, eldönthetjük, hogy egyszerű vagy összetett formot akarunk, és hogy táblákat vagy lekérdezéseket akarunk-e használni. Hagyjunk mindent alapbeállítás szerint, és lépjünk tovább a **Next** gombbal.



3.4. ábra *A Database Form varázsló első oldala*

2. A következő oldalon, ami a 3.5. ábrán látható, kiválaszthatunk egy már létező táblát, amivel dolgozni fogunk. A **Drive or Alias Name** listadobozban szerepel a DBDEMOS név, miután ezt kinyitottuk, a listában a Delphi példa adatbázis tábláit láthatjuk. Válasszuk az elsőt, az ANIMALS.DBF-et.



3.5. ábra *Tábla kiválasztása a Database Form varázslóban*

3. A harmadik oldalon kiválaszthatjuk a tábla használni kívánt mezőit. Az első példánkhoz válasszuk ki az összes mezőt a >> gombbal

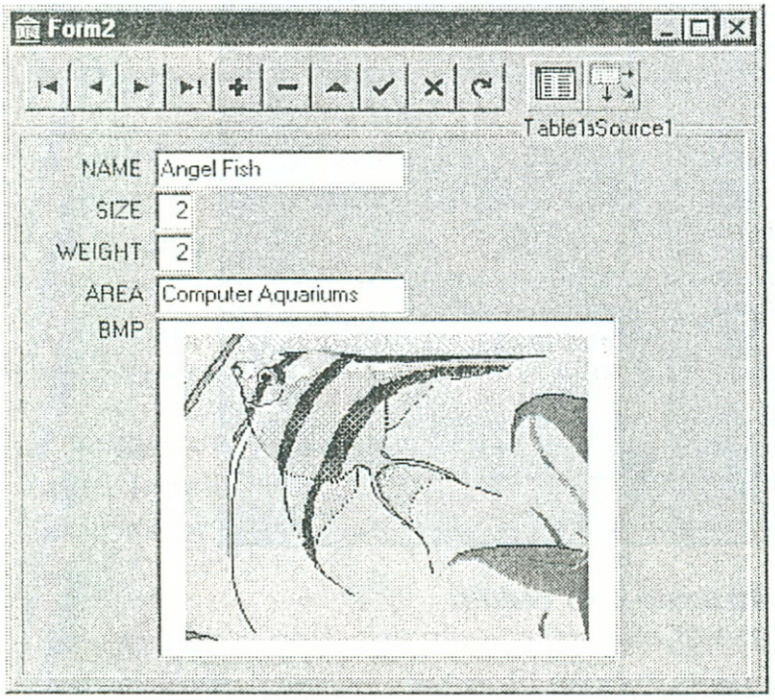
4. A következő oldalon különböző elrendezések közül válogathatunk. Ha a függőleges (vertical) választanánk, akkor a következő oldal a címkék pozícióját kérdezné. Az alapértelmezett vízszintes (horizontal) jó lesz.

5. Ez az utolsó oldal. A **Generate a Main Form** (fő form készítése) jelölőnégyzet és a **Form Only** (csak form) rádiógomb legyen kiválasztva, majd kattintsunk a **Finish** gombra.

Máris lefordíthatjuk és futtathatjuk az alkalmazást. Az eredmény egy működő adatbázis alkalmazás, ami lehetővé teszi, hogy a gombok segítségével a tételek között mozogjunk, és még egy grafikus mezőt is tartalmaz, amiben a kiválasztott állat képe látható.

A generált form általában távol áll a tökéletestől. Ebben az esetben a kép túl kicsi; máskor a kontrolok elhelyezése nem megfelelő. Szerencsére könnyedén mozgathatjuk és átméretezhetjük a komponenseket tervezés közben.

Hogy ezt egyszerűbbé tegyük, válasszuk ki a Table komponenst (a form bal felső sarkában van) és az Active tulajdonságát állítsuk True-ra. Ezután a tábla első tétele megjelenik tervezés időben is (ahogy azt a 3.6. ábrán láthatjuk). Ez nagyon hasznos, mert így példát láthatunk egy szöveges mező hosszára, és a kép méretére.



3.6. ábra *Az Adatbázis Form varázsló által generált DataExp példa formja, némi átalakítás után. Itt már be van kapcsolva a Table komponens Active tulajdonsága, hogy az adatbázis tábla adatai tervezés közben is látszódjanak*

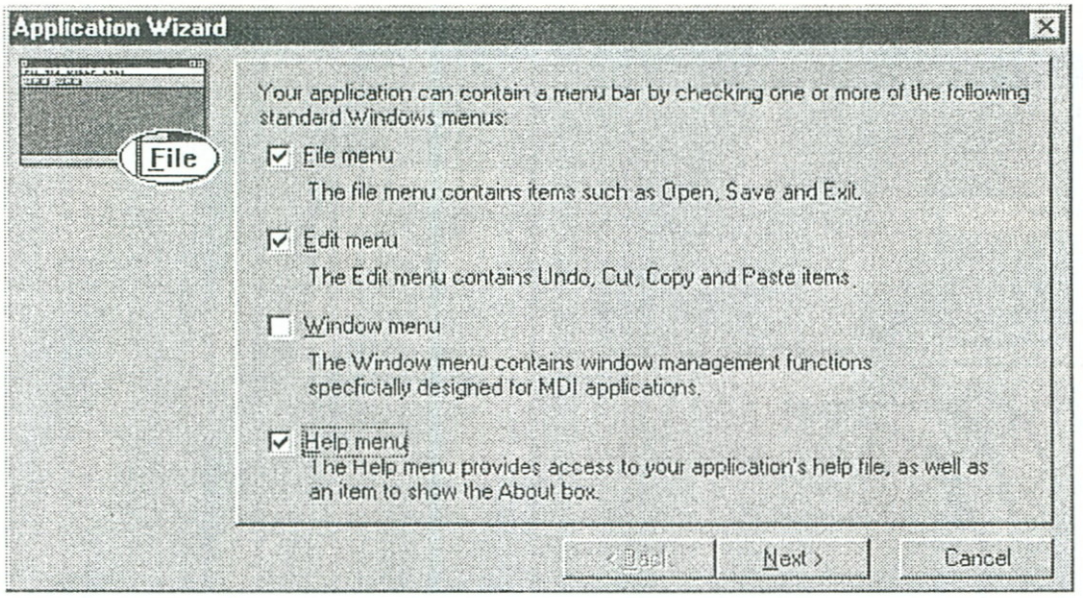
Érdemes megnézni, hogy az Adatbázis Form varázsló szinte egyáltalán nem generált Pascal kódot, eltekintve a program indításakor a tábla megnyitását végző sortól. A program tulajdonságai a Delphi adatbázissal kapcsolatos komponenseinek sokoldalúságából adódnak, amint azt a 16. fejezetben láthatjuk.

Az Alkalmazás varázsló (Application Wizard)

Másik érdekes (bár nem ennyire sokoldalú) eszköz az Application varázsló, amit az Object Repository **Projects** oldalán érhetünk el. Az Alkalmazás varázsló, a döntéseinktől függően, többféle alkalmazás elkészítését teszi lehetővé.

A varázsló első oldalán (3.7. ábra) szabványos legördülő menüket adhatunk a programhoz: File, Edit (szerkesztés), Window (ablak) és Help (súgó). Ha választunk File menüt, akkor a második oldal a program által támogatott fájl kiterjesztéseket fogja kérdezni. Ide be kell írnunk az állományok leírását, például Szövegfájlok (*.txt), és a kiterjesztést, txt (lehetőség van több kiterjesztés

megadására, és mindegyikhez saját leírás tartozhat.). Alapértelmezés szerint ezeket az értékeket fogják használni azok a File Open (megnyitás) és File Save (mentés) dialógusok, amiket az Alkalmazás varázsló ad a programhoz, ha kiválasztottuk a fájl támogatás (file support) opciót.



3.7. ábra *Az Application varázsló első oldala*

Ezután ha bármelyik legördülő menüt kiválasztottuk, akkor az Alkalmazás varázsló egy szép vizuális eszközt ad, amivel az eszköztárakat szerkeszthetjük. Sajnos ezt az eszközt nem érhetjük el külön, az editorból. Használatakor kiválasztunk egy menüt, és a menüpontoknak megfelelő számú gomb jelenik meg (de csak akkor, ha a menüt az Alkalmazás varázsló első oldalán kiválasztottuk).

Új gomb hozzáadásához válasszunk egyet a jobb oldali grafikus listapanelből, és nyomjuk meg az **Insert** gombot. Az új gomb a kis háromszög által jelzett helyre kerül. Ezt a jelet elmozgathatjuk, ha egy már létező elemre kattintunk az eszköztáron. Ugyanez a jel mutatja azt a gombot, amit el akarunk távolítani az eszköztárból.

Amikor az eszköztár készen van, átléphetünk az utolsó oldalra. Itt beállíthatunk néhány opciót, mint például az MDI támogatás, a státuszsor hozzáadása, és a tippek engedélyezése. Itt adhatunk nevet az új alkalmazásnak, és határozhatjuk meg a forrásfájlok mappáját. Az alkalmazás neve lehet hosszú, de nem lehetnek benne elválasztó karakterek (általában formailag helyes Pascal azonosítónak kell lennie), és az alkalmazás mappájának már léteznie kell. Ahhoz, hogy a projekt állományait új mappába tegyük, válasszuk ki a **Browse** gombot, írjuk be az új útvonalat, és a párbeszédablak rá fog kérdezni, hogy szeretnénk-e új mappát létrehozni.

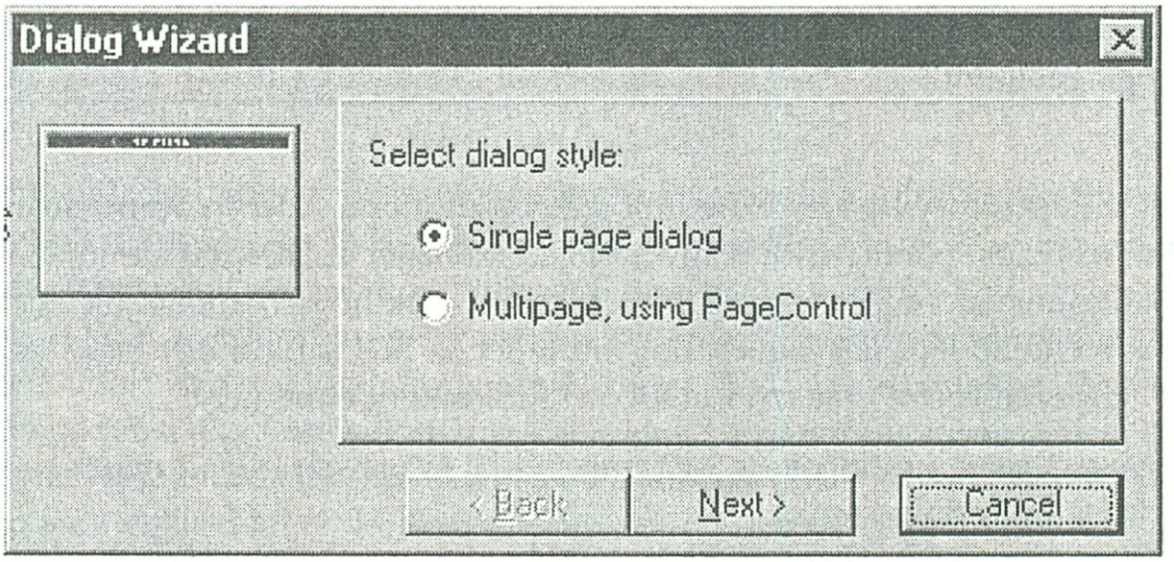
Bár kissé hiányos, és ráférne még a fejlesztés, a Delphi Alkalmazás varázsló új alkalmazások első verzióinak elkészítésekor sokkal hasznosabb, mint az előre elkészített alkalmazás sablonok. Az egyik legnagyobb előnye, hogy mi készíthetjük el a saját eszköztárunkat, a másik hatalmas előnye, hogy sokkal több kódot (és több megjegyzést) ír, mint a hasonló sablonok. A varázsló hátránya viszont, hogy egy formos alkalmazásokat készít, MDI támogatása korlátozott, mivel nem készít gyermekablakot, ezen kívül Névjegyet sem.

A Párbeszédablak varázsló (Dialog Wizard)

A Delphi Dialog varázslója egy egyszerű varázsló, ami leginkább csak példaként szerepel, a saját forrásával együtt. Elméletileg ennek a varázslónak a kódjából megtanulhatjuk, hogy hogyan készíthetünk saját varázslókat – ettől azonban még nyugodtan használhatjuk a Párbeszédablak varázslót kétféle párbeszédablak, az egyszerű és a többboldalas elkészítésére. Ezek a Windows 95 PageControl komponensére épülnek (3.8. ábra).

Ha egyszerű párbeszédablakot választunk, a varázsló a harmadik oldalra ugrik, ahol beállíthatjuk a gombok elhelyezkedését. Ha többboldalasat választunk, akkor egy köztes oldal jelenik meg, ahol a különböző fülek feliratait adhatjuk meg.

Ez a varázsló csak egy alternatíva az Object Repositoryban lévő sablonok mellett, előnye, hogy közvetlenül szerkeszthetjük a PageControl füleinek feliratát. A párbeszédablakokkal bővebben a 11., a PageControl komponenssel pedig a 12. fejezetben fogunk foglalkozni.



3.8. ábra

A Dialog varázsló első oldala

Az Object Repository testreszabása

Mivel új varázsló készítése nem egyszerű feladat, ezért az Object Repository testre szabásának leggyakrabban használt módszere az, hogy új projekteket, formokat, és adatmodulokat adunk hozzá sablonnak. Létrehozhatunk új oldalakat, és elrendezhetjük az elemeket (a **New**, és az „aktuális projekt" oldalak nem tartoznak ide).

Új alkalmazás sablon felvétele

Új sablon elhelyezése a Delphi Object Repository-jába ugyanolyan egyszerű, mint felhasználni egy már meglévő sablont egy új alkalmazás elkezdéséhez. Amikor elkészül egy működő alkalmazás, és azt szeretnénk két vagy több hasonló program fejlesztésekor kiinduló állapotként felhasználni, akkor elmenthetjük az aktuális állapotot egy sablonba, ezzel felkészítve a további felhasználásra.

Megjegyzés

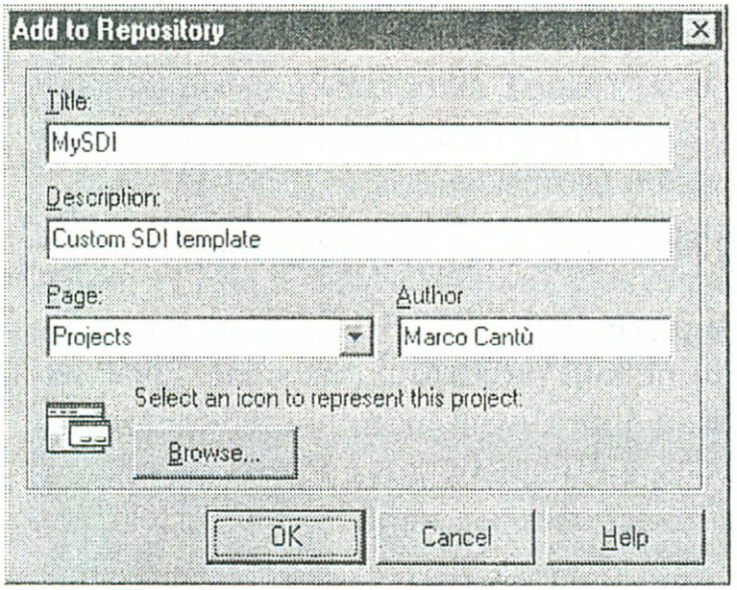
Bár a Borland mindent objektumnak hív, amit az Object Repositoryba betehetünk, objektumorientált nézőpontból ez távol áll az igazságtól, ezért, a későbbi felhasználásra lemezre mentett sémákat sablonoknak hívjuk. Az alkalmazássémák egyáltalán nem kapcsolódnak objektumokhoz vagy osztályokhoz, csak az új projekt könyvtárába másolódnak. Az Object Repository ugyan sokkal jobban hangzik, mint a Browse Gallery, de a formöröklötetés képességén kívül nincs sok újdonság ebben az eszközben.

Projektet a Repository-ba helyezhetünk a **Project** menü **Add to Repository** (tárolóba helyezés) parancsával, vagy a Project Manager ablak saját menüjének megfelelő pontjával. Nagyon egyszerű példaként, csak hogy bemutassuk az eljárást, a következő lépések leírják, hogyan lehet az SDI sablon egy kissé módosított változatát (erről volt szó korábban) sablonként elmenteni:

1. Nyissuk meg a módosított SdiTemp példát (vagy bármilyen másik projektet, amin dolgozunk).
2. Válasszuk ki a **Project** menü **Add to Repository** parancsát (vagy a Project Manager ablak eszközsoráról az **Add Project to Repository** parancsot.)
3. Az **Add to Repository** párbeszédablakon (3.9. ábra) írjunk be egy címet és egy leírást az új projekt sablonról, és adjuk meg a szerző nevét.

Választhatunk még ikont az új sablonnak, vagy elfogadhatjuk a felkínáltat. Végül válasszuk ki azt a Repository oldalt, amire a projektünket tenni szeretnénk.

4. Kattintsunk az **OK**-ra, és az új sablon a Delphi Object Repository része lesz.



3.9. ábra *Az Add to Repository dialógust új alkalmazás sablon nevének, és leírásának meghatározására használhatjuk*

Mostantól ha megnyitjuk az Object Repositoryt, mindig meg fog jelenni a saját sablonunk, ha később úgy döntünk, hogy ezt a sablont többé már nem használjuk, eltávolíthatjuk. A projekt sablonokat arra is használhatjuk, hogy egy meglévő projektről másolatot készítsünk, és az eredeti elmentése után folytathassuk a munkát.

Ennek azonban van egyszerűbb módja is: másoljuk a forrásállományokat egy másik mappába, és nyissunk meg egy új projektet. Amikor a forrásállományokat átmásoljuk, az ablakok képernyő pozícióját tároló DSK állományt hagyjuk ki. A DSK az editorban megnyitott állományok listáját is tartalmazza, teljes elérési úttal, ez azt jelenti, hogy amint megnyitjuk az új projektet, és elkezdünk dolgozni, a régi forrásállományokat fogjuk szerkeszteni, de az új verzió állományait fogjuk fordítani (mivel a project manager relatív útvonalakat tárol). Ez nagyon meglepő lehet, különösen, amikor a formon, vagy a kódon végzett változtatásaink látszólag teljesen hatástalanok. A DSK állomány letörlésével, vagy át nem másolásával elkerülhetjük ezt a problémát.

Az üres projekt sablon

Amikor új alkalmazásba kezdünk, mindig megjelenik egy üres form is. Ha valamilyen form objektummal, vagy varázslóval szeretnénk dolgozni, akkor általában ez nem lesz számunkra megfelelő. Ennek a problémának megoldására létrehozhatunk egy Üres Projekt sablont.

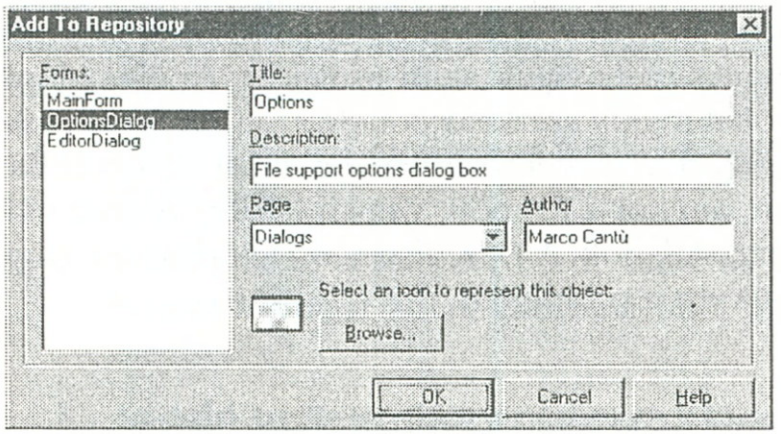
Az ehhez szükséges lépések a következők:

1. Készítsünk egy új projektet.
2. Távolítsuk el a projekt egyetlen formját.
3. Adjuk ezt a projektet a sablonokhoz, Üres Projekt néven.

Ha az Object Repositoryból ezt a projektet választjuk, akkor az két dolog miatt is jó. Egyrészt van egy form nélküli projektünk, másrészt kiválaszthatjuk azt a mappát, ahová a sablon állományait másolni szeretnénk. Van azonban a dolognak egy hátránya is, a **File** menü **Save Project As** parancsával más nevet kell adnunk a projektnek, mert alapbeállításban a sablon nevét fogja használni.

Új form sablonok hozzáadása

Új form sablonokat éppen úgy adhatunk hozzá az Object Repositoryhoz, amint azt új projekt sablonoknál láthattuk. Keressük meg a hozzáadni kívánt formot, kattintsunk rajta a jobb egérgombbal, és a SpeedMenüből válasszuk ki az **Add to Repository** parancsot. A megjelenő párbeszédablakban (3.10. ábra) kiválaszthatjuk, hogy az aktuális projekt melyik formját szeretnénk a Repositoryban elhelyezni, ezután, a megszokott módon beállíthatjuk a nevét, a leírását, a szerző nevét, az oldalt és az ikont. Miután ezeket az beállítottuk kattintsunk az **OK**-ra, és a form máris az Object Repository megfelelő oldalára kerül.

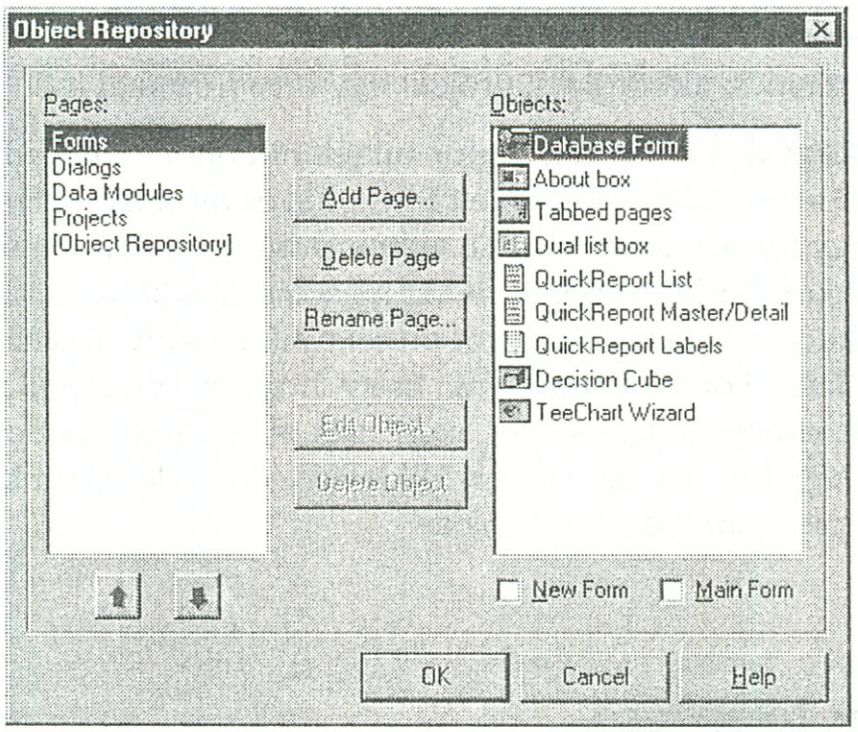


3.10. ábra *A form elmentése sablonként, az Add Form to Repository párbeszédablak használatával*

Ez a megoldás javasolt akkor, ha összetett formokat szeretnénk más alkalmazások, vagy programozók számára elérhetővé tenni. Ezzel a formunkat önmagában fel lehet használni, le lehet másolni, lehet örököltetni belőle. Emiatt a form Repositoryba tétele sokkal rugalmasabb, mint a projekteké, mivel azokat csak új alkalmazások kiindulópontjaként lehet felhasználni.

Az Object Repository opciói

További testre szabáshoz a **Tools** (Eszközök) menü **Repository** parancsával előhívható Object Repository párbeszédablakot (3.11. ábra) használhatjuk. Ezt a párbeszédablakot könnyű használni: bal oldalt szerepelnek a Repository oldalak, jobbra pedig az oldalakon található elemek, sablonok és varázslók láthatók. Ebben a dialógusban az elemeket mozgathatjuk az oldalak között, felvehetünk újakat, és törölhetjük a régieket.



3.11. ábra *Az Object Repository Options párbeszédablak*

A három oldalakkal kapcsolatos, és az oldal lista alatti két, nyilakat ábrázoló, gombbal az Object Repository struktúráját módosíthatjuk, új oldalakat készíthetünk, meglévőket átnevezhetünk vagy letörölhetünk, és megváltoztathatjuk a sorrendjüket. Ezek a műveletek az Object Repository néhány fülét érintik, míg más fülek rögzítettek.

Az Object Repository beállításainak fontos eleme az alapértékek használata:

- Az objektum lista alatti **New Form** jelölőnégyzet segítségével a kiválasztott formot vagy varázslót alapértelmezetté tehetjük, vagyis új form létrehozásánál (a **File** menü **New Form** parancsa) a rendszer ezt fogja használni. Csak egyetlen Object Repository elem lehet alapérték; ezt az ikonjára helyezett különleges szimbólum jelöli.
- A **Main Form** jelölőnégyzet azt jelzi, hogy az új alkalmazás (a **File** menü **New Application** parancsa) fő formját a kiválasztott form vagy varázsló segítségével szeretnénk elkészíteni, ha nincsen különleges beállítás az új projektre vonatkozóan (lásd a következő pontot). Az aktuális főformot egy másik különleges szimbólum jelöli.
- A **New Project** jelölőnégyzet csak akkor él, ha projekt objektumot választottunk, használatával megadhatjuk, hogy a Delphi mit használjon a **File** menü **New Application** parancsánál. Az Új projektnek is megvan a saját különleges jele.

Ha nincsen kijelölt Új projekt, akkor a Delphi az új projekteket a fő form beállítások alapján készíti, ha ez sincs, akkor az új projekt egy üres formmal indul.

Amikor az Object Repositoryval dolgozunk, akkor tulajdonképpen a Delphi főkönyvtárának OBJREPOS almappájába elmentett formokat és modulokat látjuk. Amikor új projektet vagy formot adunk hozzá, akkor az is ebbe a mappába kerül elmentésre. Ugyanakkor, ha közvetlenül, tehát másolás nélkül, felhasználunk egy objektumot, akkor oda jutunk, hogy a projektünk néhány állománya ebben a mappában lesz. Fontos megérteni, hogy hogyan működik a Repository, mert ha egy, itt elhelyezett projektet, vagy más objektumot módosítani akarunk, akkor a legjobb, ha az eredeti állományokon dolgozunk, anélkül, hogy oda-vissza másolgatnánk az adatokat.

Új varázsló DLL-ek telepítése

A varázslók általában kétféle helyről jöhetnek. Lehetnek komponensek, vagy csomagok részei, ebben az esetben ugyanúgy kell telepíteni, mint egy komponenst vagy csomagot. Más varázslókat önálló DLL állományok formájában terjesztenek. Ebben az esetben a DLL nevét hozzá kell adni a Windows Registryhez, a következő kulcs alá:

```
Software\Borland\Delphi 3.0\Experts
```

Csak hozzá kell adni egy új karakterláncot ez alá, akármilyen nevet választhatunk (az ugyanis lényegtelen), és a szöveghez írjuk be a varázsló DLL-jének elérési útját és nevét. A már meglévő bejegyzésekben megnézhetjük, hogy hogyan kell ezeket megadni.

Hogyan tovább?

Ebben a rövid fejezetben láthattuk, hogy hogyan kezdhetünk új fejlesztést a Delphi sablonjai és varázslói segítségével. Használhatjuk az előre elkészített sablonok egyikét, elindíthatjuk az Adatbázis Form varázslót, vagy akármelyik másik varázslót, hogy gyorsabban elkezdhessük új alkalmazás, form vagy más objektum fejlesztését.

A könyv többi részében nagyon ritkán fogunk varázslókat vagy sablonokat használni, mivel az Adatbázis Form varázsló kivételével a többi eszközt általában csak nagyon egyszerű alkalmazásokat készít, amit gyakorlott Delphi programozóként percek alatt összehozhatunk. Kezdőknek a varázslók és a sablonok gyorsabb elindulást jelentenek a fejlesztés során, de ki akar örökre kezdő maradni?

Az első rész további fejezeteiben tehát az Object Pascal nyelvet fogjuk tanulmányozni, és felfedezzük a Delphi VCL (vizuális komponens könyvtár) összetevőit. Ezután egy valódi projektbe kezdünk, és megismerjük a varázslók által generált, vagy a sablonokban tárolt kódot.

4. FEJEZET

A Pascal nyelv

E fejezet tartalma:

- Típusok, változók és konstansok
- Delphi adattípusok
- Stringkezelés
- Programozási stílus
- Feltételes kifejezések és ciklusok
- Eljárások és függvények
- Hivatkozások és nyitott sorrendű paraméterek
- Eljárási típusok

A Pascal nyelv

A Pascal nyelv a többi programozási nyelvtől különböző, egyedi jellemzőkkel rendelkezik. Megszületését Dr. Niklaus Wirth professzornak köszönhetjük, aki 1972-ben teremtette meg ezt az új alapokon nyugvó nyelvet, melyet a francia Blaise Pascal filozófus után nevezett el. Egyszerű szerkezete és ötletes megvalósítása miatt hamar a világ egyik legkedveltebb és legsokrétűbben használható programozási nyelvévé vált.

Ez a népszerűség nem kis részben a Borland cég 1985-ben piacra dobott Turbo Pascal sorozatának volt köszönhető. A nyelvnek eme változata PC-s platformon különösen közkedveltté tette a Pascalt, ezt az egyszerű, mégis nagy lehetőségeket rejtő eszközt. Manapság a Delphi sikere nyomán reneszánszát éli a Pascal, és annak objektumorientált változata, az Object Pascal.

A Pascal alapjainak megírásakor már sok programozási nyelv létezett, de közülük csak kevés volt széleskörűen elterjedt. A nyelv kialakításakor a fő hangsúly a rendezett, élesen elhatárolt adattípusokon, és az áttekinthető, strukturált programozási környezet megteremtésén volt. Ez a könyv nem foglalkozik behatóbban magával a Pascal programozási nyelvvel, de ebben a fejezetben számos olyan információt találhatunk, amely segíthet egyes alapvető programozási szabályok Pascal nyelvben való alkalmazásában.

Akár kevés, akár sok tapasztalattal rendelkezünk a Pascal nyelvvel kapcsolatban, érdemes ezt a fejezetet az elejétől a végéig elolvasni. Az összes terítékre kerülő példa a Delphi környezetből való, amely sok szempontból a Pascal nyelv kiterjesztésének nevezhető. Az Object Pascal objektumorientált eszközeivel a következő fejezetben találkozhatunk.

Típusok, változók és konstansok

Az eredeti Pascal nyelv olyan elgondolásokon alapult, amelyek napjainkban szinte kötelezővé váltak a modern programozási nyelvekben. Az egyik ilyen az adattípus, amely meghatározza, hogy egy változó milyen értékeket vehet fel, és milyen műveletek végezhetőek el rajta. A típus fogalma a Pascalban sokkal kötöttebb, mint a C nyelvben, ahol az aritmetikai adattípusok szinte helyettesíthetők egymással, és jelentősen fejlettebb az eredeti BASIC nyelvnél, ahol semmilyen típusfogalom nem található.

Változók

A Pascal nyelv megköveteli a változók deklarálását azok első használata előtt. Ezzel egy időben nem csak a majdan használni kívánt változó nevét kell megadni, hanem annak típusát is. Nézzünk erre néhány példát:

```
var
  Value: Integer;
  IsCorrect: Boolean;
  A, B: Char;
```

A `var` kulcsszót a forráskódon belül több helyen is elhelyezhetjük: az eljárások vagy a függvények kezdetén, ha a változót rutinon belüli, lokális változóként, illetve egy uniton (egységen) belül, ha globális változóként akarjuk felhasználni (erről bővebben a következő fejezet hatáskörrel foglalkozó részeiben olvashatunk). A kulcsszó után a változók nevei és típusai következnek, kettősponttal elválasztva. Egy sorban több változót is megadhatunk, ahogyan ezt a példa utolsó sorában is láthatjuk.

A változó meghatározása után csak az azon a típuson alkalmazható műveletek végezhetők a változón. Például, használhatunk logikai (Boolean) változót igazsághalmaz kiértékelésre és egész (Integer) típust számok meghatározására, de nincs lehetőségünk egész és logikai típusok közötti műveletek végzésére (nem úgy, mint a C nyelvben).

Egyszerű értékadásokat használva leírhatjuk a következőt:

```
Value := 10;
IsCorrect := True;
```

A következő példa viszont hibás, mert a két változó típusa különböző:

```
Value := IsCorrect; // error
```

Tipp

A Delphi hibaüzenetek jelentős fejlesztésen mentek át. A 16 bites verziónál sokkal részletesebb lettek, és képesek tanácsok és figyelmeztetések megjelenítésére, amely a hibakeresés eredményességét nagyban növelheti. Lehetőleg engedélyezzük ezeket a Project Options párbeszédablak **Compiler** oldalán keresztül.

Amennyiben ilyen vagy ehhez hasonló értékadást próbálunk meg lefordítani Delphi alatt, úgy az „Incompatible types: 'Integer' and 'Boolean'." hibajelzést kapjuk. Általában az ehhez hasonló hibák programozási hibák, hiszen nincs értelme egy integer típushoz True (igaz) vagy False (hamis) érték rendelésének Az ilyen hibaüzenetekért nem a Delphit kell hibáztatni, hiszen az legtöbbször a saját figyelmetlenségünk következménye.

Ennek ellenére természetesen lehetséges a különböző típusok közötti átalakítás. Egyes esetekben ez automatikusan megtörténik, de legtöbbször beépített rendszerutasításokat kell hívni annak elvégzésére.

A Delphi forráskódban a változó meghatározásakor megadhatjuk annak kezdeti értékét is. Például:

```
var
  Value: Integer = 10;
  Correct: Boolean = True;
```

Ezt a technikát csak a globális változókra alkalmazhatjuk, eljárások vagy függvények lokális változóinál nem.

Konstansok

A Pascal nyelvben lehetőség van konstansok megadására is. Ezek olyan speciális „változók", amelyeknek értéke az egész program futása alatt változatlan marad. Egy konstans létrehozásához csak annak nevét és értékét kell megadni, a megfelelő típus hozzárendelése a fordító feladata. Lássunk néhány példát a konstans létrehozására:

```
const
  Thousand = 1000;
  Pi = 3.14;
  AuthorName = 'Marco Cantù';
```

Mint ahogyan arról már szó esett, a Delphi a konstans típusát annak értéke alapján automatikusan dönti el, így a fenti példában szereplő Thousand konstans a SmallInts típust veszi fel, mivel az a legszűkebb típus, amely azt az értéket tartalmazni tudja. Ha viszont azt szeretnénk, hogy egy konstans az általunk meghatározott típust vegye fel, akkor azt a következő módon érhetjük el:

```
const
  Thousand: Integer = 1000;
```

A konstans létrehozásakor a fordító dönti el, hogy annak számára memóriablokkot foglal le, bemásolja a konstans értékét, majd arra hivatkozik, vagy a program minden egyes helyére automatikusan befűzi a konstans értékét, ahol azt használjuk. Ez utóbbinak főleg egyszerűbb konstansok esetén van értelme.

Figyelem!

A Delphi 16 bites verziójában a program futása közben megváltoztathatjuk a konstansok értékét, mintha azok egyszerű változók lennének. A 32 bites verzió a visszafelé való kompatibilitási okokból engedélyezi ezt, ha a fordító $J kapcsolója, vagy a Project Options párbeszédablak **Compiler** oldalán található **Assignable typed constans** kapcsoló be van kapcsolva. Ha lehet, ne használjuk ki ezt az egyszerű trükköt, mert automatikusan megszünteti a fordító összes, a konstansok optimalizálásra való törekvését.

String erőforrás (resource string) konstansok

Amikor egy string konstanst határozunk meg, akkor a

```
const
  AuthorName = 'Marco Cantù';
```

helyett a következőt írhatjuk:

```
resourcestring
  AuthorName = 'Marco Cantù';
```

Mind a két esetben konstanst hozunk létre, amelynek értéke nem változik a program futása alatt. A különbség ezeknek a programon belüli implementálásában rejlik. Az így létrehozott szövegkonstans a program erőforrásai között foglal helyet, egy, erre a célra kialakított string táblázatban (a 27. fejezet foglalkozik mélyebben a Windows erőforrásaival. Általánosságban az mondható el, hogy az erőforrások használatával előnyösebben használható ki a memória, és könnyebb a programot más nyelvre adaptálni).

Ezt a lehetőséget, amely a Delphi 3-ban jelenik meg először, használja ki a következő példánk, amely egy egyszerű nyomógombot hoz létre:

```
resourcestring
  AuthorName = 'Marco Cantù';
  BookName = 'Mastering Delphi 3';
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage (BookName + #13 + AuthorName);
end;
```

A kimenetként megjelenő stringek két külön sorban foglalnak helyet, mert a programrészletben a két string között egy újsor karakter található. Ez utóbbi a #13 karakter típusú konstanssal van ábrázolva.

Ha ennek a programnak a forráskódját egy szövegszerkesztővel tekintjük meg (a mellékelt CD-n is található egy), akkor egy igen érdekes jelenséget figyelhetünk meg: a stringek pontosan úgy néznek ki, mintha a program futása közben látnánk őket. Többek között ezzel a jelentséggel is foglalkozunk majd a 27. fejezetben.

Delphi adattípusok

A Pascal nyelvben számos előre meghatározott adattípus létezik, amelyeket három különböző csoportba sorolhatunk: ordinal (sorszámozott), real (valós), és string (karakterlánc, szöveg). A következő szakaszokban a sorszámozott és a valós adattípusokat tárgyaljuk meg, míg a string típussal kapcsolatos részek a fejezet végén találhatók. Itt ugyancsak megismerkedhetünk olyan adattípusokkal , amelyeket nem közvetlenül a fordító, hanem az egyes Delphi könyvtárak határoznak meg.

Delphi programok írása közben használhatunk egy speciális, nem meghatározott adattípust, amelyet Variant-nak (választhatónak) nevezünk. Ez a kifejezés egy szabályos típusellenőrzés nélküli adattípust takar, először a Delphi 2-ben jelent meg az OLE alkalmazások kezelésével kapcsolatban. A fejezet végén olvashatunk arról, hogy a választható adattípus segítségével milyen módon lehet leküzdeni a Pascal szigorú típusellenőrzési szabályait, illetve a 22. fejezetben behatóbban foglalkozunk az OLE alkalmazások használatával.

Sorszámozott típusok (ordinal types)

A sorszámozott típusok halmazába azon elemek sorolhatók, amelyek valamilyen szempont alapján sorrendbe állíthatók. Két tetszőleges, sorszámozott típusba tartozó elemről nem csak azt mondhatjuk meg, hogy melyik a nagyobb, hanem azt is egyértelműen meghatározhatjuk a halmazon értelmezett sorrend alapján, hogy mely elemek fogják közre közvetlenül az adott elemet, illetve meghatározhatjuk a sor legnagyobb, és legkisebb elemét.

A három legfontosabb sorszámozott típus az Integer (egész szám), Boolean (logikai) és a Char (karakter). Ezeken kívül több, hasonló jellemzőkkel bíró adattípus is létezik, amelyek csak belső reprezentációjukban és értékkészletükben különböznek. A következő táblázat a számokra alkalmazható sorszámozott adattípusokat tartalmazza:

Méret	Előjeles	Előjel nélküi
8 bit	ShortInt	Byte
16 bit	SmallInt	Word
32 bit	LongInt	
16/32 bit	Integer	Cardinal

Láthatjuk, hogy ezek a típusok a számok különféle reprezentációihoz tartoznak, attól függően, hogy az adott számot hány biten akarjuk ábrázolni, és hogy fel kívánunk-e használni egy bitet az előjel jelölésére. Az előjellel rendelkező típusokkal pozitív és negatív értékeket is ábrázolhatunk, de ennek fejében csak kisebb pozitív és negatív értékkészletet használhatunk, hiszen egy bitet az előjel jelzésére foglalunk le. Az egyes típusokkal ábrázolható számokat a következő szakaszban található Range példán keresztül mutatjuk be.

Az utolsó, 16/32 bit-tel jelölt csoport értékei különbözőképpen vannak ábrázolva a Delphi 16 bites és 32 bites változatai alatt. Az Integer és a Cardinal a legtöbbet használt típus, mert ezek közvetlenül megfelelnek a CPU-ban használatos számok adatszerkezetének.

A numerikus Boolean típusokat (melyek nem azonosak a logikai típussal) bátran sorolhatjuk a legritkábban használt típusok közé. A Windows API

függvényei speciális logikai értékek használat teszik szükségessé, ezek a ByteBool, WordBool és a LongBool típusok.

Új!

A Delphi 3-ban a Visual Basic-kel és az OLE Automationnel való kompatibilitási okokból a ByteBool, WordBool és a LongBool logikai típusok az igaz értéket a -1-gyel , a hamis értéket pedig a 0-val jelölik. A Boolean logikai adattípus változatlan maradt (igaz: 1, hamis: 0). Ezért, ha közvetlen típusmegfeleltetést tartalmazó Delphi 2 programot használunk Delphi 3 alatt, akkor az elég nagy valószínűséggel hibát fog okozni.

Végül még azt is meg kell még említeni, hogy a karakterek ábrázolására két típust használhatunk a Delphin belül: az ANSIChar és a WideChar típusokat. Az előbbi típus a szabványos ANSI 8 bites karakterek, míg az utóbbi a most elterjedőben lévő 16 bites Unicode karakterek megjelenítésére alkalmas. Legtöbbször általában az ANSIChar típusnak megfelelő Char típust fogjuk használni, arról azonban sohase feledkezzünk meg, hogy az első 256 Unicode karakter teljesen megegyezik az ANSI karakterekkel.

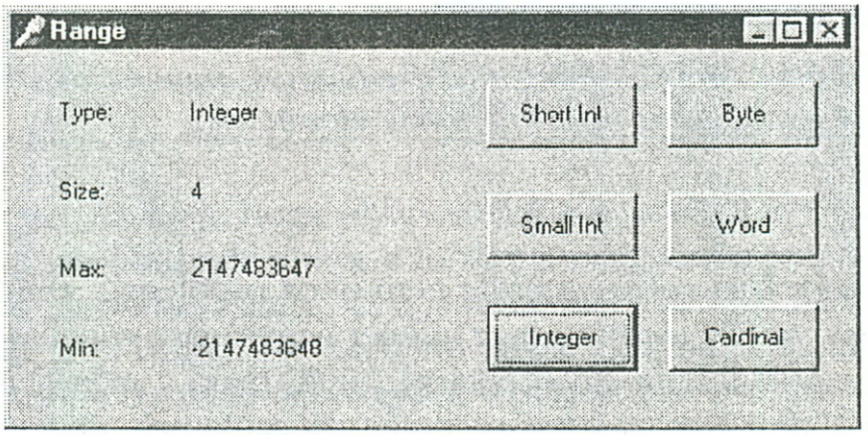
A Range példaprogram

A sorszámozott típusok különböző értékkészleteinek és határainak bemutatásához egy kitűnő eszköz a Delphiben írt Range példaprogram. Ezt a mellékelt CD-ről lemásolhatjuk és lefordíthatjuk minden különösebb nehézség nélkül. A program futtatása után a 4.1. ábrához hasonló eredményt kell kapnunk.

A Range példaprogram egy egyszerű formot hoz létre, amely hat gombon (mindegyik egy-egy sorszámozott típus után van elnevezve) és néhány információs mezőn kívül semmi mást nem tartalmaz (4.1. ábra). A mezők egy része statikus szöveget tartalmaz, míg a többi egy-egy gomb lenyomása után az adott gombhoz rendelt típus értékeit adja vissza.

Különböző mezők mutatják az adat típusát, a felhasznált bitek számát, és a típusban eltárolható legkisebb és legnagyobb értéket. Minden egyes gombnak megvan a maga OnClick eseménykezelője, amely a gombhoz tartozó három mező értékét számolja ki. Erre azért van szükség, mert az értékek kiszámításának módja típusonként különbözik. Például az Integer gomb (BtnInteger) OnClick esemény által aktivált programrészlete a következőképpen néz ki:

```
procedure TFormRange.BtnIntegerClick(Sender: TObject);
begin
  LabelType.Caption := 'Integer';
  LabelSize.Caption := IntToStr (SizeOf (Integer));
  LabelMax.Caption := IntToStr (High (Integer));
  LabelMin.Caption := IntToStr (Low (Integer));
end;
```

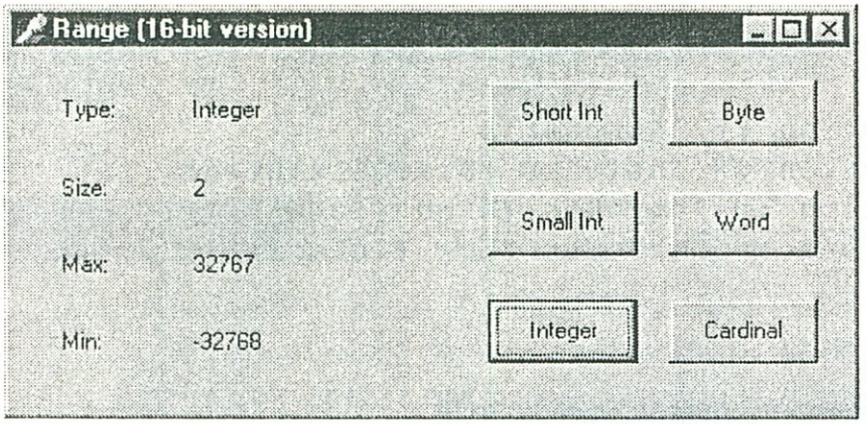


4.1. ábra *A Range példaprogram információkat jelenít meg az általában használt adattípusokról (jelen esetben az egészekről)*

Amennyiben már rendelkezünk némi tapasztalattal a Delphi programozásának terén, úgy bátran elemezzük a példaprogram és működésének menetét. Kezdőknek elég, ha a három függvény: SizeOf, High és Low kezelését sajátítják el. Ez utóbbi kettő mindig a hívott típusnak megfelelő típusú eredményt ad vissza (jelen esetben Integert), amíg a SizeOf mindig Integer típusú végeredményt produkál.A programban mindhárom függvény visszatérési értékét stringgé alakítjuk az IntToStr eljárás segítségével, majd bemásoljuk őket a nekik megfelelő mezőbe.

A többi gomb által alkalmazott eljárások nagyrészt megegyeznek a fent említettel. Az egyetlen, de lényeges különbség a különböző függvények által visszaadott eredmények típusa. A 4.2. ábra a példaprogram a 16-bites, Windows 95 alatt futó változatának értékeit mutatja (ennek a forráskódja a Range16 név alatt található meg a könyvhöz mellékelt CD-n). A 4.1. és a 4.2. ábrát összehasonlítva tisztán látszik a 16 és a 32 bites Integer adattípusok közötti különbség.

Az Integer típus nagysága a programot futtató gép hardverétől és operációs rendszerétől is függ: a 16 bites Windows alatt az Integer 2 byte hosszú, míg a 32 bites Windows alatt négy. Ez az oka annak, hogy különböző eredményeket kapunk a Range példaprogram újrafordítása után.



4.2. ábra *A Range 16 bites verziójának futási eredménye, ugyancsak az integerekkel kapcsolatos információkat jeleníti meg*

Az Integer típus kétféle ábrázolása egészen addig nem okoz problémát, amíg a program nem alkalmaz semmiféle megkötést az integer nagyságára vonatkozóan. Előfordulhat, hogy egy fájlba az egyik fajtával írunk, majd a másikkal szeretnénk olvasni belőle. Ekkor igen érdekes eredmények születnek, de legtöbbször nem az, amit szeretnénk. Ilyen esetekben tanácsos egy platformfüggetlen ábrázolási módot választani (mint a Longint vagy SmallInt). Matematikai számolásokhoz és általános számábrázolásokhoz használjuk a platform alapvető számábrázolását – azaz az Integer típust – mert ez az, amit a CPU a leggyorsabban kezel. A egész számok kezelésénél is mindig az Integer típusra kell először gondolnunk, más típust lehetőleg csak akkor használjunk, ha nyomós okunk van rá.

Sorszámozott típusokon használható eljárások és rutinok

Néhány beépített rendszerrutin (amelyeket a Pascal nyelv és a Delphi rendszerkönyvtára ad meg) sorszámozott típuson is alkalmazható. Ezeket a 4.1. táblázat tartalmazza. A C++-ban járatos programozók észrevehetik, hogy az Inc két verziója, egy és két paraméterrel, a ++ és a += operátoroknak felelnek meg. Ugyanez mondható el a Dec eljárásról is.

4.1 táblázat

A sorszámozott típusokon használható beépített eljárások

Eljárás	Hatása
Dec	A paraméterként megadott változó értéket eggyel, vagy a második paraméter megadása esetén annak értékével csökkenti.
Inc	A paraméterként megadott változó értéket eggyel, vagy a második paraméter megadása esetén annak értékével növeli.

4.1 táblázat
A sorszámozott típusokon használható beépített eljárások (folytatás)

Eljárás	Hatása
Odd	True (igaz) értéket ad vissza, ha argumentuma páratlan szám.
Pred	A sorszámozott típusú érték megelőzőjét adja vissza.
Ord	A sorszámozott típus sorszámával tér vissza.
Succ	A sorszámozott típusú érték rákövetkezőjét adja vissza.
Low	A sorszámozott típusban tárolható legkisebb értékkel tér vissza.
High	A sorszámozott típusban tárolható legnagyobb értékkel tér vissza.

Jegyezzük meg, hogy olyan esetekben, amikor ezeket a rutinokat konstansokon alkalmazzuk, akkor a fordító a programba annak előre kiszámolt értékét helyettesíti be. Például, ha egy forrásban a következő sort hívjuk meg: High(X), ahol az X változót Integerként adjuk meg, úgy a fordító a kifejezést az Integer típusban tárolható legnagyobb értékkel helyettesíti.

Valós típusok

A valós típusokkal a lebegőpontos számokat lehet különböző formátumokban ábrázolni. A legkisebb ezek közül a Single, amely 4 bájton ábrázolja a számokat. A Double típus ezzel szemben 8 bájt, míg az Extended 10 bájt hosszú. Ezek a különböző hosszúságú típusok más és más precizitású lebegőpontos számábrázolást tesznek lehetővé, az IEEE nemzetközi lebegőpontos szabványnak megfelelően. Ezek mindegyike a matematikai koprocesszor által közvetlenül is támogatott, így alkalmazásukkal a lehető leggyorsabb kód létrehozása érhető el.

Figyelem!

A Real típus csak a régebbi Pascal forráskódokkal való kompatibilitási célok miatt maradt benne a Delphi 3-ban, ezért lehetőleg ne használjuk, ugyanis ennek 6 bájtos szerkezete nem képezi részét az IEEE nemzetközi lebegőpontos számábrázolási szabványának, ezért a Delphi esetleges későbbi verzióiból eltűnhet ez a típus.

Ugyancsak meg kell említeni még két különleges adattípust: a nagyon nagy számok ábrázolására használható 8 bájt hosszú Comp (nem található meg a Delphi 16 bites verziójában), és a Currency, amely nem más, mint egy fixpontos érték négy tizedesjeggyel, a Comp 64 bites megfelelője. Ahogyan azt a neve is mutatja, ez utóbbi főleg pénzügyi alkalmazásokban és átalakításoknál használható eredményesen.

Mivel a valós típusokon nincsenek a High, Low és az Ord függvények értelmezve, ezért nem írhatunk a Range-hez hasonló programot a valós adattípusok felhasználásával. A valós típusokkal (elméletileg) végtelen sok szám ábrázolható, míg a sorszámozott típusokkal véges számú. Ezekből kifolyólag könnyen meghatározható egy ábécén belüli betű helye, de nincsen értelme a 7143,1562 szám sorszámát keresni. Attól függetlenül, hogy könnyedén megállapíthatjuk, hogy melyik valós számnak van nagyobb értéke, lehetetlen egyértelműen meghatározni, hogy egy adott valós szám előtt hány másik található (amelyet egyébként az Ord függvény ad meg).

A valós típusoknak eléggé korlátozott szerepük van a felhasználói felület létrehozásába (a Windows oldalán), de ettől függetlenül a Delphi teljeskörű támogatását élvezik, beleértve az adatbáziskezelést is. Az IEEE szabványok alapján felépülő Object Pascal nyelv a számok ábrázolása miatt a legtöbb matematikai számításokat végző programmal kompatíbilis. Ezeket Delphi által nyújtott matematikai függvényeken keresztül érhetjük el (további információkat a Delphi Súgójában kaphatunk erről a témáról).

Megjegyzés

A Delphi tartalmaz egy Math unitot, amelyen keresztül speciális matematikai függvények, közöttük a trigonometrikus (mint például az ArcCosh függvény), üzleti (mint az InterestPayment függvény) és statisztikai függvények (mint a MeanAndStdDev eljárás) érhetőek el. Ezek közül sok eléggé furán hangzik, mint például a MomentSkewKurtosis eljárás (funkciójának megfejtését az Olvasóra bízom).

Dátum és idő

A Delphi a dátum és az idő tárolására valós típusokat alkalmaz. Ezek pontosságának növeléséhez egy speciális adattípust, a TDateTime-ot használ. Ez is egy lebegőpontos szám, hiszen elég nagynak kell lennie az évek, hónapok, napok, órák, percek, másodpercek és századmásodpercek egyetlen változóban való tárolására. A dátumok az 1899 december 30 óta eltelt napok számában van megadva (negatív érték 1899 előtti éveket jelöl) a típus egész értékein. Az idő pedig századmásodpercben, a típus tizedesjegy részében található.

A TDateTime nem egy, a fordító által közvetlenül értelmezett, hanem a System unitban (egységben) előre meghatározott típus:

```
type
  TDateTime = type Double;
```

A TDateTime típus használata meglehetősen könnyű, mert nagyon sok, ezen a típuson alkalmazható függvényt találhatunk a Delphi beépített eljárásai között. Ezeket a függvényeket a 4.2 táblázat tartalmazza.

4.2. táblázat

A TDateTime típuson alkalmazható eljárások listája

Eljárás	Leírás
Now	Az aktuális dátumot és időt egy TDateTime típusú értékben adja vissza.
Date	Az aktuális dátummal tér vissza.
Time	Az aktuális idővel tér vissza.
DateTimeToStr	A dátumot és az időt stringgé konvertálja, az alapértelmezett formátum alapján. Pontosabb átalakításhoz használjuk helyette a FormatDateTime függvényt.
DateTimeToString	A dátumot és az időt egy string bufferbe helyezi, az alapértelmezett formátumban.
DateToStr	A TDateTime típus dátum részét stringgé konvertálja.
TimeToStr	A TDateTime típus idő részét stringgé konvertálja.
FormatDateTime	Az időt és dátumot összetett, általunk meghatározható formátumúvá alakítja.
StrToDateTime	Stringből TDateTime típusú értéket hoz létre, illetve hibaértékkel tér vissza, ha a string formátuma nem megfelelő.
StrToDate	Dátumot tartalmazó stringet TDateTime formátumúvá alakít.
StrToTime	Időt tartalmazó stringet TdateTime formátumúvá alakít.
DayOfWeek	A TdateTime értékként megadott stringből visszaadja, hogy a megadott nap a hétnek hanyadik napja.
DecodeDate	A dátumból visszaadja az évet, hónapot és napot.
DecodeTime	Az időpotból az órát, percet és a másodpercet adja vissza.
EncodeDate	A megadott évből, hónapból és napból TDateTime értéket képez.
EncodeTime	A megadott órából, percből és másodpercből TDateTime értéket képez.

Az előbb felsorolt eljárások működésének megértése érdekében nézzük meg a TimeNow nevű példaprogramot. A program formja csak egyetlen gombot és egy ListBox elemet tartalmaz. A program indítása után azonnal kiszámolja, majd megjeleníti az aktuális dátumot és az időt, a formon található gomb minden egyes megnyomása után pedig az indítás óta letelt idő jelenik meg.

A form OnCreate eseményéhez tartozó programrészlet:

```
procedure TFormTimeNow.FormCreate(Sender: TObject);
begin
  StartTime := Now;
  ListBox1.Items.Add (TimeToStr (StartTime));
  ListBox1.Items.Add (DateToStr (StartTime));
  ListBox1.Items.Add ('Press button for elapsed time');
end;
```

Az első sorban található a Now eljárás meghívása, ami az aktuális dátummal és idővel tér vissza. Ezt az értéket a StartTime globális változóban tároljuk, amelyet a következőképpen határoztunk meg:

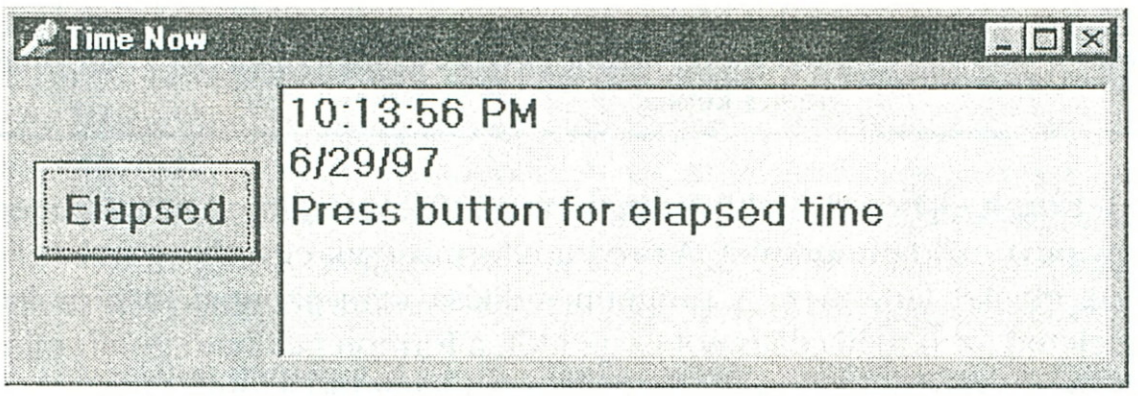
```
var
  FormTimeNow: TFormTimeNow;
  StartTime: TDateTime;
```

Ezek közül csak az utóbbit kell nekünk létrehozni, ugyanis az elsőt a Delphi már tartalmazza a következőképpen:

```
var
  Form1: TForm1;
```

Ha a form nevét megváltoztatjuk, akkor annak a meghatározása is automatikusan módosul. A globális változók használata ebben az esetben nem a legjobb megoldás, helyette inkább a form osztályához tartozó különálló mezőt fogunk majd alkalmazni. Ezzel bővebben a következő fejezetben fogunk foglalkozni, ahol az osztályokkal és egyéb objektumorientált programozási fogásokkal fogunk majd megismerkedni.

A következő három kifejezés három elemet ad a ListBox elemhez, ami hatására a 4.3. ábrán látható eredményt kapjuk. Az első sor a TDateTime érték idő részét tartalmazza, stringgé alakítva, a második a dátumot, hasonló módon, míg az utolsó rész egy egyszerű szöveget tartalmaz. A harmadik string tartalma akkor változik meg, ha az Elapsed feliratú gombra kattintunk.



4.3. ábra *Így néz ki a TimeNow példaprogram indításának pillanatában*

```
procedure TFormTimeNow.ButtonElapsedClick(Sender:
TObject);
var
  StopTime: TDateTime;
begin
  StopTime := Now;
  ListBox1.Items [2] := FormatDateTime ('hh:nn:ss',
      StopTime - StartTime);
end;
```

Ez a kódrészlet lekérdezi az aktuális időt, majd kiszámolja a program indítása óta eltelt időtartamot. Mivel nekünk egy másik eseménykezelővel beállított értékkel kell dolgoznunk, szükséges egy globális változó létrehozása és használata. A következő fejezetben viszont látni fogjuk, hogy létezik egy jobb, osztályokon alapuló megoldás is.

Megjegyzés

A kód, amely a harmadik sztring értékét megváltoztatja, a kettes indexet használja. Ez onnan származik, hogy a listában szereplő elemek sorszámozása nulla alapú, azaz az első elemnek a száma 0, a másodiké 1, és így tovább. Erről többet a tömböknél tudhatunk meg.

A TimeToStr és DateToStr függvények hívásán kívül használható még a FormatDateTime eljárás is, mint ahogy a fent említett példában látható (lásd a Delphi Súgóját a paraméterek megadásával kapcsolatban). Ne hagyjuk figyelmen kívül azt sem, hogy az idő és dátumformátumok a Windows beállításaitól is nagy mértékben függnek. A Delphi ezeket az értékeket a rendszerből olvassa ki, majd átmásolja őket különböző, SysUtils unit által meghatározott globális változókba. Ezek közül néhány:

```
DateSeparator: Char;
ShortDateFormat: string;
LongDateFormat: string;
TimeSeparator: Char;
TimeAMString: string;
TimePMString: string;
ShortTimeFormat: string;
LongTimeFormat: string;
ShortMonthNames: array[1..12] of string;
LongMonthNames: array[1..12] of string;
ShortDayNames: array[1..7] of string;
LongDayNames: array[1..7] of string;
```

Több globális változó Currency és más lebegőpontos számábrázolásokat használ. Ezeket a Delphi Help fájljának „Currency and date/time variables" nevű menüpontja alatt találhatjuk meg.

Új!

A Delphi 3 egy új DateTimePicker nevű komponenst is tartalmaz, amiről bővebben a 11. fejezetben olvashatunk.

Különleges Windows-típusok

Az eddig látott adattípusok zömmel a Pascal nyelv alaptípusai közé tartoznak. A Delphi nyelvben ezeken kívül a Windowsban használatos adattípusokkal is találkozhatunk. Ezek nem a beépített eljárásokhoz tartoznak, hanem a Windows könyvtáraihoz, amelyek többek között új típusokat (például DWORD vagy UINT), számos mutatótípust és sok más újdonságot is tartalmaznak.

A Windows adattípusok közül kiemelten fontos szerepet töltenek be az úgynevezett handle-k (kezelők). Ezekre a THandle névvel hivatkozhatunk a Windows unitban a következőképpen:

```
type
  THandle = Integer;
```

A handle adattípusokat úgy használjuk, mintha azok számok lennének, holott nem teljesen azok; a Windows-on belül a handle-k belső adatszerkezetekre való hivatkozások. Például, mikor ablakokkal vagy Delphi formokkal dolgozunk, akkor az ablakot ilyen handle-n keresztül érhetjük el. Az ablak létrehozásakor a rendszer egy számot rendel az ablakhoz, és attól fogva ezzel a számmal hivatkozhatunk rá amikor mozgatjuk, átméretezzük, vagy egyéb változtatást eszközölünk rajta.

A legtöbb Windows API függvénynél a handle-t első paraméterként kell megadni. Ez azonban nem csak az ablakon alkalmazott eljárásokra vonatkozik: más Windows API eljárások is megkövetelik a saját handle-t első paraméterként. Többféle handle típus is létezik, mint például menü, bitmap (bittérkép) és így tovább.

Más szóval, a handle egy belső kód, amely a rendszer megfelelő elemeinek és objektumainak azonosítására szolgál, beleértve az ablakokat, ikonokat, bittérképeket, memória blokkokat, kurzorokat, fontokat, menüket és még sok minden mást. A Delphi nyelvben a handle-ket nagyon ritkán kell direkt formában

használni, hiszen azok belső programozási felületek és objektumok között helyezkednek el. Használatukat csak akkor kell közvetlen módon kezdeményezni, ha olyan Windows API eljárásokat hívunk, amelyeket a Delphi programozási nyelv nem támogat.

Megjegyzés

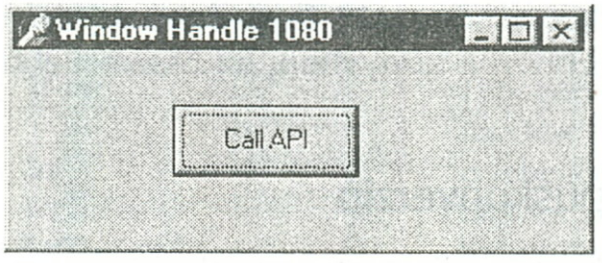
A handle-k nagysága különbözik a 16 bites és a 32 bites Windowsban. Ugyanez igaz természetesen a 16 bites és a 32 bites Delphire is. Ha erre nem figyelünk, komoly problémákat és fejtörést okozhatunk magunknak. A legtöbb esetben a handle-k nagysága nem lényeges.

Ablakkezelők (handle)

A leírás teljessé tételéhez elengedhetetlen egy rövid, a témához kapcsolódó példa. A WHandle példaprogram egyszerű formjához egyetlen gomb tartozik. A forráskódon belül a következőképpen határozzuk meg a formhoz rendelt OnCreate és a gombhoz kapcsolódó OnClick eseményeket:

```
object FormWHandle: TFormWHandle
  Caption = 'Window Handle'
  OnCreate = FormCreate
  object BtnCallAPI: TButton
    Caption = 'Call API'
    OnClick = BtnCallAPIClick
  end
end
```

A form létrehozása után a program megszerzi az ahhoz tartozó ablak handlejét a form Handle tulajdonságain keresztül. Ezután a handle értéket az IntToStr eljárás stringgé alakítja, majd a form Caption (Felirat) tulajdonságához konkatenálja, ahogyan azt a 4.4 ábrán láthatjuk:



4.4. ábra *A WHandle példaprogram egy form handle-jét mutatja be. A futtatás minden alkalmával más és más értéket kapunk*

```
procedure TFormWHandle.FormCreate(Sender: TObject);
begin
  Caption := Caption + ' ' + IntToStr (Handle);
end;
```

Megjegyzés

Ebben az eljárásban egyszerűen hivatkozhatunk a form fejlécére és a Handle tulajdonságára, mivel ezek a form saját metódusai, amellyel ugyanabban az osztályban található egyéb tulajdonságokat és metódusokat érhetünk el közvetlenül.

A program többszöri futtatása után mindig különböző handle értéket kapunk, ugyanis ezt az értéket a Windows határozza meg, és adja át az alkalmazásnak. Ezért a handle-k értéke soha sincs előre, illetve az alkalmazás által meghatározva, értéküket a rendszertől kapják, és ez az érték minden egyes alkalommal más és más lehet.

Amikor a gombra kattintunk, akkor az alkalmazás egyszerűen meghívja a SetWindowText nevű Windows API függvényt, amely a paraméterben megadott ablak szövegét vagy feliratát változtatja meg. Egészen pontosan, az API függvény első paramétere az a handle, amely azt az ablakot azonosítja, amelyet változtatni akarunk:

```
procedure TFormWHandle.BtnCallAPIClick(Sender: TObject);
begin
  SetWindowText (Handle, 'Hi');
end;
```

Jegyezzük meg, hogy ennek a programrészletnek ugyanaz a hatása, mint az előző eseménykezelőnek, amely az ablakban található szöveget változtatja meg a form Caption jellemzőjének átállításával. Ebben az esetben az API függvény hívása teljesen értelmetlen, mivel rendelkezésünkre áll egy jóval egyszerűbb módszer Delphi alatt. Sajnos azonban néhány API függvénynek nincs Delphi megfelelője, ahogyan ezt később ebben a könyvben, az összetettebb példák esetében látni fogjuk.

Típusmegfeleltetés (casting) és típuskonverzió

Ahogyan azt korábban láthattuk, akkor is értékül adhatunk egy változónak egy másikat, ha azok különböző típusúak. Amennyiben erre szükség van, úgy két megoldás közül választhatunk. Az első ilyen a típusmegfeleltetés (casting),

amely egyszerű eljárásjelölést használ, a kimeneti változó nevének a felhasználásával:

```
var
  N: Integer;
  C: Char;
  B: Boolean;
begin
  N := Integer ('X');
  C := Char (N);
  B := Boolean (0);
```

Csak megegyező nagyságú adattípusok között végezhető típusmegfeleltetés. Általában biztonságosnak mondható a sorszámozott és a valós adattípusok egymás közötti megfeleltetése. Ugyancsak lehetséges pointer típusok és objektumok közötti megfeleltetés létrehozása, egészen addig, amíg tudjuk, hogy mit csinálunk.

A megfeleltetés ennek ellenére, egy veszélyes programozási technika, mert segítségével olyan adatokat rendelhetünk egy változóhoz, amelyek teljesen más rendeltetésű értékeket tartalmaznak. Mivel az adattípusok belső szerkezete eltérő, így elég könnyen hozhatunk létre nehezen felderíthető hibákat. Ebből a szempontból tanácsos a típusmegfeleltetések használatának elkerülése.

A második módszer a típuskonverzió. A különböző típusok közötti átalakításokra használható eljárásokat a 4.3. tábla tartalmazza. Ezek közül néhány olyan típusok átalakítását végzi, amelyet mi csak a következő szakaszokban tárgyalunk. Vegyük észre, hogy ez a tábla nem tartalmazza a speciális típusoknál (mint például a TDateTime és hozzá hasonlók), vagy adatok (például a Format vagy a FormatFloat) formázásánál használatos rutinokat.

4.3. táblázat

A típuskonverzióra használatos rendszereljárások

Eljárás	Leírás
Chr	Sorszámozott számot ANSI karakterré alakít át.
Ord	Sorszámozott típusú értéket alakít annak sorszámává.
Round	Lebegőpontos értéket egész típusú értékké konvertál, kerekítéssel.
Trunc	Lebegőpontos értéket egész típusú értékké alakít át, a tizedestört érték levágásával.
Int	A lebegőpontos érték egész részével tér vissza.
IntToStr	Számot stringgé konvertál.
IntToHex	Számból stringet alakít, a tizenhatos számrendszernek megfelelően.
StrToInt	Stringből számot állít elő. Hibával tér vissza, ha a stringből nem képezhető érvényes egész szám.

4.3. táblázat

A típuskonverzióra használatos rendszereljárások (folytatás)

Eljárás	Leírás
StrToIntDef	Stringet számmá konvertál. Ha nem sikerül, akkor az alapértelmezett értéket adja vissza.
Val	Stringet számmá alakít át (hagyományos Turbo Pascal eljárás. Csak kompatibilitási okokból maradt meg).
Str	Számból a formázási paramétereknek megfelelő stringet képez (hagyományos Turbo Pascal eljárás, csak kompatibilitási okokból maradt meg).
StrPas	Nullával lezárt stringet Pascal-típusúvá alakít. Ez az eljárás automatikusan végrehajtódik a 32 bites Delphiben az AnsiString eljárás hívásakor.
StrPCopy	Pascal-típusú stringből nullával lezárt stringet képez. A 32 bites Delphiben ugyanezt a PChar eljárással érhetjük el (lásd a stringekről szóló részt ennek a fejezetnek a végén).
StrPLCopy	A megadott Pascal-típusú string egy részét egy nullával lezárt stringbe másolja.
FloatToDecimal	Lebegőpontos értéket rekordba konvertál, ami tartalmazza a kitevőt, a mantisszát és az előjelet (exponent, digits, sign).
FloatToStr	Lebegőpontos számot az alapértelmezett formázási paraméternek megfelelő stringgé alakít.
FloatToStrF	Lebegőpontos számot a paraméterként megadott formázási stílus szerint stringgé alakít.
FloatToText	Lebegőpontos számot string bufferbe másol, a megadott formázás szerint.
FloatToTextFmt	Hasonlóan az előző eljáráshoz, ez is a lebegőpontos értéket egy string bufferben helyezi el, a megadott formázási rendszer szerint.
StrToFloat	Pascal stringet lebegőpontos számmá alakít.
TextToFloat	Nullával lezárt stringet lebegőpontos értékké alakít.

A Variant típus

Az OLE teljes körű támogatásának érdekében a Delphiben megtalálható a Variant adattípus. Bár elsősorban a Variant típus speciális felhasználási területeire helyezzük a fő hangsúlyt, itt az általános szemszögből tekintjük át ennek az adattípusnak a működését. A Variant típus valójában egy, az egész nyelven keresztül érezhető hatással rendelkezik, így nem is lehet meglepő, hogy a Delphi könyvtárak gyakran használják, még ott is, ahol az OLE eljárásokat és elemeket egyáltalán nem is veszik azt igénybe.

Általában a Variant változókat felhasználhatjuk tetszőleges adatok tárolására, különleges műveletek elvégzésére és típusátalakításokra is. Jegyezzük meg,

hogy ez szöges ellentétben áll a Pascal nyelv eredeti elképzeléseivel és a kiforrott, optimális és tökéletesnek tekintett programozási technikákkal. A Variant típusa és értéke mindig a program futása közben dől el. Ilyenkor kivételesen nem kell a fordító típusellenőrzésére figyelnünk. Egészében véve, a Variant változókat tartalmazó kódrészleteket tekinthetjük értelmezett (interpretált) kódrészletnek is, hiszen a legtöbb műveletet csak futás közben értékelhetjük ki, amely erősen befolyásolja a program sebességét.

Most, hogy tisztában vagyunk a Variant típus veszélyeivel, lássuk, hogy mire is jó ez. Alapjában véve, miután létrehoztunk egy Variant változót a következőképpen:

```
var
  V: Variant;
```

különböző típusokhoz tartozó értékeket rendelhetünk hozzá:

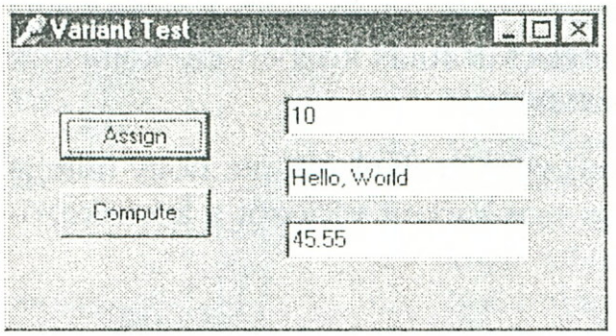
```
V := 10;
V := 'Hello, World';
V := 45.55;
```

Miután rendelkezünk egy Variant értékkel, azt bármilyen, vele kompatíbilis vagy inkompatibilis adattípusba átmásolhatjuk, mivel a Delphi automatikusan elvégzi az átalakítást helyettünk, ha tudja. Ha nem, akkor hibát (kivételt) generál. Azt meg kell jegyezni, hogy a Variant az értéken kívül a típus adatait is tárolja, ezzel is segítve különleges, futás közbeni műveletek elvégzését. Ez utóbbiak, bár hasznosak lehetnek, lassú és nem túl biztonságos programot eredményeznek.

Vegyük például a következő programrészletet (a VariTest nevű példaprogramból való, ami ugyancsak megtalálható a mellékelt CD-n), amely nem más, mint az előbbi program okosabb változata. Egy új formon három szerkeszthető dobozt (edit box) és egy pár gombot helyezünk el, majd az első gombhoz hozzárendeljük a következő, OnClick esemény által aktivált programrészletet:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  V: Variant;
begin
  V := 10;
  Edit1.Text := V;
  V := 'Hello, World';
  Edit2.Text := V;
  V := 45.55;
  Edit3.Text := V;
end;
```

Vicces, nem? Amellett, hogy egy stringet rendelünk a Variant típushoz az első körben, később integer és lebegőpontos értéket is adunk neki, és mindezek ellenére a dolog működik, mint ahogyan azt a 4.5 ábra mutatja.



4.5. ábra *A VariTest példaprogram az Assign nyomógomb (Button1) megnyomása után*

És a furcsaságoknak még nincs vége! Azt is el kell árulnunk, hogy a változtatható típus segítségével még értékek kiszámolására is lehetőség nyílik. Ezt mutatja be a következő, a második gombhoz rendelt, programrészlet:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  V: Variant;
  N: Integer;
begin
  V := Edit1.Text;
  N := Integer(V) * 2;
  V := N;
  Edit1.Text := V;
end;
```

Az ehhez hasonló programrészletek írása még finoman szólva is életveszélyes. Ha az első mezőbe számot adunk, akkor minden a lehető legnagyobb rendben működik. De adjunk csak meg valami mást, és rögtön egy kivétel (exception) váltódik ki. Ezért, és a hasonló szituációk miatt csak nagyon nyomós indok esetén alkalmazzunk Variant típust a programunkban. Ha tehetjük, inkább használjuk helyette az eredeti Pascal adattípusokat és adatmegközelítési technikákat. Ebből kifolyólag, a könyvben kevés helyen találkozhatunk majd a Variant típussal.

Variant típusok a Delphiben

A Delphi többek között tartalmazza a TVarData rekord típust is, amely a Variant típussal megegyező szerkezettel rendelkezik. Ennek használatával a változtatható típusból kinyerhető annak aktuális tartalma. A TVarData tartalmazza a

Variant típusát (VType által jelölt), néhány fenntartott mezőt, és végül, a Variant típusban tárolt értéket.

A Variant lehetséges típusai az OLE típusától függenek. A következő lista a nekik megfelelő konstansokat tartalmazza:

```
varEmpty      varNull       varSmallint    varInteger
varSingle     varDouble     varCurrency    varDate
varOleStr     varDispatch   varError       varBoolean
varVariant    varUnknown    varString      varTypeMask
varArray      varByRef
```

Ezen típusoknak teljes leírása a Delphi súgójában található meg, a „Values in variants" címszó alatt.

Ugyancsak sok, a változatokat használó eljárás létezik, amelyekkel sajátos típusátalakításokat végezhetünk el, vagy információkat kérdezhetünk le egy-egy magadott Variant típusról (lásd a VarType függvényt). A legtöbb ilyen típusátalakítás és hozzárendelő eljárás a Variant típusok használata közben automatikusan hívódik meg. Egyéb, Variant típusokat támogató eljárásokat a Súgóban a „Variant support routines" címszó alatt találhatunk még.

A Variant lassú!

Jegyezzük meg, hogy a Variant típus használata nem csak akkor lassítja jelentős mértékben a program futását, amikor típusátalakítást végzünk, hanem akkor is, amikor két ilyen változóban tárolt integert adunk össze; körülbelül annyira, mintha Visual Basicben írtuk volna meg. Hogy tiszta képet nyerjünk a módszerek sebességkülönbségéről, nézzük meg a mellékelt CD-n található VSpeed példaprogramot, amely ehhez hasonló ciklusokat futtat le (a while ciklusról bővebben szólunk még ebben a fejezetben, egy kicsivel később):

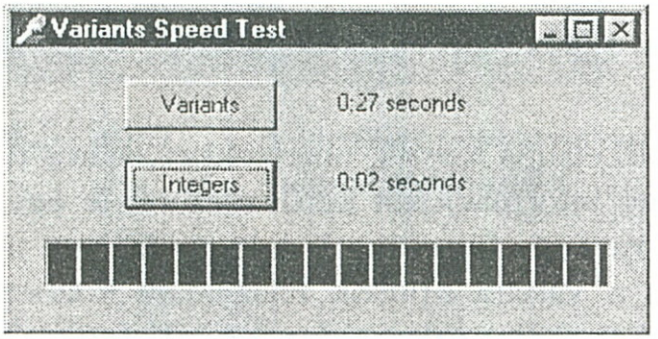
```
while n1 <5000000 do
begin
  n2 := n2 + n1;
  Inc (n1);
end;
```

Ebben a ciklusban, ahogyan azt a forráskódra pillantva láthatjuk, néhány belső művelet is található, amely a folyamatsáv (progress bar) frissítését, és az alkalmazás működőképességét és időzítését garantálja. Ezek legtöbbje ebben a pillanatban számunkra egyáltalán nem lényeges, de mivel már tárgyaltuk a TDateTime típust, ezért nézzük meg a ciklus futási idejét mérő programrészletet:

```
begin
  time1 := Now;
  // loop...
  time2 := Now;
```

```
    Label1.Caption := FormatDateTime (
      'n:ss', Time2-Time1) + ' seconds';
end;
```

Ahogyan azt láthatjuk, a FormatDateTime függvényt használjuk fel, s annak is csak a perceket és másodperceket jelölő részét (n:ss). A futási sebességek közötti különbség akkora, hogy azt mérés nélkül is észre lehet venni (4.6. ábra). Igazság szerint, ezek az idők nagy mértékben függenek a futtató gép típusától és kiépítésétől is, de egymáshoz viszonyított arányuk általában megegyezik.



4.6. ábra *Ugyanannak az algoritmusnak más és más a futási sebessége, ha csak integerekre, illetve ha csak Variant típusra van megírva (az aktuális értékek nagy mértékben függhetnek a futtató gép típusától), ahogyan azt a VSpeed példaprogram bemutatja*

Felhasználó által meghatározott adattípusok

A típus fogalmának bevezetésén kívül a Pascal nyelv másik nagyszerű ötlete volt a felhasználó által meghatározható típusok létrehozásának biztosítása. Ezáltal a programozó maga hozhatja létre a neki tetsző típust, a rendelkezésre álló felsorolás, sorozat, rekord, megszámlálható, halmaz típusok, és pointerek által. A legfontosabb felhasználó által meghatározott adattípus a class, azaz magyarul az osztály. Ez utóbbival a következő fejezetben foglalkozunk.

Ezeknek a típusoknak saját nevet adhatunk, amit később egyszerűen hozzárendelhetünk egy-egy változóhoz. Amikor nevet adunk egy ilyen típusnak, akkor egy külön szakaszt kell létrehozni a forráskódon belül a következőképpen:

```
type
  // subrange definition
  Uppercase = 'A'..'Z';

  // array definition
  Temperatures = array [1..24] of Integer;

  // record definition
  Date = record
    Month: Byte;
```

```
    Day: Byte;
    Year: Integer;
end;

// enumerated type definition
Colors = (Red, Yellow, Green, Cyan, Blue, Violet);

// set definition
Letters = set of Char;
```

A felhasználó által meghatározott típust a változóhoz közvetlenül is hozzá lehet rendelni, külön típusmeghatározás nélkül. Erre mutat egy példát a következő programrészlet:

```
var
  DecemberTemperature: array [1..31] of Byte;
  ColorCode: array [Red..Violet] of Word;
  Palette: set of Colors;
```

Figyelem!

Általában véve, lehetőleg kerüljük a névtelen típusok használatát, ahogyan azt a fenti példában láthatjuk, ugyanis azokat nem adhatjuk át paraméterként a rutinoknak és nem is hozhatunk létre velük egyező típusú változókat. A Pascal típusok közötti kompatibilitási szabálya a típusok nevein alapul és nem azok fajtáján. Két teljesen megegyező típus is lehet egymással inkompatibilis. Szokjuk meg, hogy minden egyes esetben létre kell hozni egy adattípust, ha egy bonyolultabb változóra van szükségünk. Ilyen esetekben ne törődjünk az idővel, mert előbb utóbb rá fogunk jönni, hogy ez rengeteg energiát és munkát takarít meg nekünk.

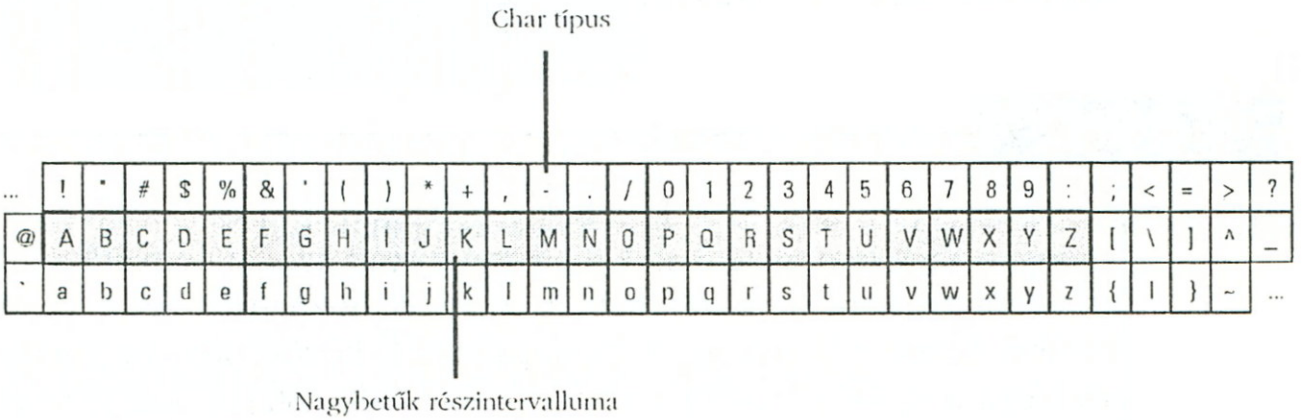
Joggal merülhet fel a kérdés, hogy mit is jelentenek ezek a típusmeghatározások. Valószínűleg rendelkezünk már elég tapasztalattal ezekkel kapcsolatban, mégis fussunk át gyorsan a főbb jellemzőkön, kiemelve azokat, amelyek más programozási nyelvekkel szemben eltérően működnek, majd tekintsünk át néhány Delphi példát, amelyből megtanulhatjuk, hogy milyen eszközökkel, és hogyan lehet az egyes típusokhoz tartozó információkat dinamikusan kezelni.

A részintervallum típusok (subrange types)

A részintervallum típus egy nagyobb intervallumon belüli kisebb intervallumot határoz meg (innen a neve). Meghatározhatjuk vele például az integer típuson belül az 1-től 10-ig, vagy 100-tól 1000-ig futó részintervallumot, vagy a Char típus egy részintervallumát a következőképpen:

```
type
  Ten: 1..10;
  OddRange: 100..1000;
  Uppercase = 'A'..'Z';
```

Ez utóbbi részintervallumot a 4.7 ábrán láthatjuk. A részintervallum típusának meghatározásánál nem szükséges a bővebb intervallum típusának meghatározása, elég, ha abból két konstans értéket választunk ki, amelyeknek sorszámozott típusúnak kell lenniük, és végeredményként is sorszámozott típust kapunk.



4.7. ábra

A részintervallum reprezentációja

A részintervallum típus meghatározásakor szabályosnak tekinthető az intervallumon belüli érték hozzárendelése a változóhoz, így a következő példa helyes:

```
var
  UppLetter: UpperCase;
begin
  UppLetter := 'F';
```

Ez viszont nem:

```
var
  UppLetter: UpperCase;
begin
  UppLetter := 'e'; // compile-error
```

A fenti programrészlet beírása és lefordítása után a „Constant expression violates subrange bounds", vagyis „Konstans kifejezés megsérti a részintervallum határait" hibaüzenetet kapjuk. Ellenben ha a következő programrészletet fordítjuk le:

```
var
  UppLetter: Uppercase;
  Letter: Char;
begin
  Letter :='e';
  UppLetter := Letter;
```

akkor azt a Delphi gond nélkül lefordítja. Futása közben, amennyiben a **Range checking** (intervallum ellenőrzés) fordítási opciót engedélyeztük (ezt a Project Options párbeszédablak **Compiler** oldalán keresztül), úgy a „Range check error" hibaüzenetet kapjuk.

Tipp

Kapcsoljuk be ezt a fordítási kapcsolót a program fejlesztése közben, és csak a végleges verzió elkészítésekor kapcsoljuk ki, ezzel ugyanis a program egy picivel gyorsabb és kisebb lesz. Ugyanez vonatkozik a futás közbeni ellenőrzési lehetőségekre, mint például a túlcsordulás és a veremellenőrzés. Ez utóbbiakat lehetőleg még a végleges verzióban is hagyjuk meg.

Felsorolt típusok

A felsorolt típus egy másik, sokat használt és hasznos, a felhasználó által meghatározott típus. Itt ahelyett, hogy egy részintervallum alsó és felső határát adnánk meg, egyenként felsoroljuk az összes lehetséges értéket. A felsorolás típus tehát nem más, mint lehetséges értékek listával való meghatározása. Íme néhány példa:

```
type
  Colors = (Red, Yellow, Green, Cyan, Blue, Violet);
  Suit = (Club, Diamond, Heart, Spade);
```

Az egyes értékek sorrendje egyértelműen meghatározott, az első érték indexe nulla. Ha az Ord függvényt alkalmazzuk a felsorolt típusra, akkor egy, nulla, vagy annál nagyobb értékeket kapunk vissza. Például az Ord(Diamond) az 1 értékkel tér vissza.

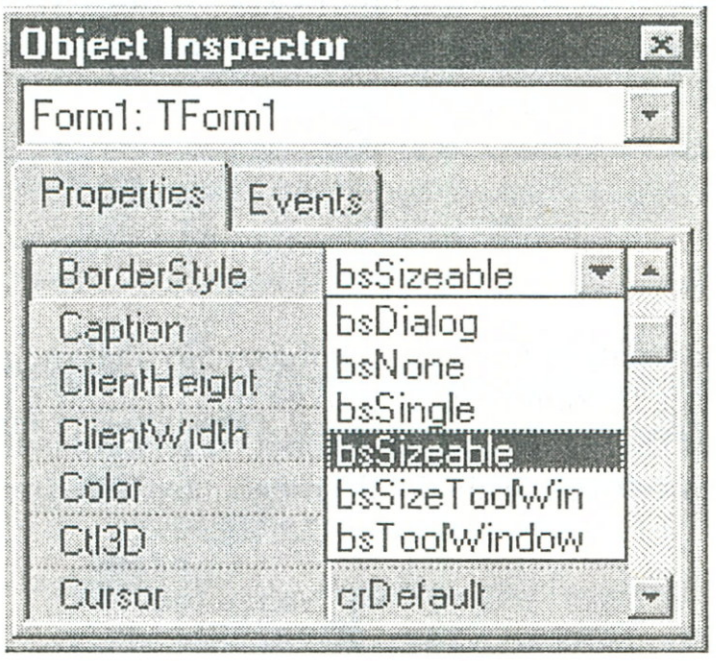
Megjegyzés

A megszámlálható típusoknak más és más ábrázolásai lehetnek. Alapállapotban a Delphi a 8 bites ábrázolást használja, kivéve, ha 256-nál több értékkel dolgozunk. Ez utóbbi esetben a 16 bites ábrázolást használja. Létezik egy 32 bites ábrázolási mód is, ami a C és C++ könyvtárakkal való kompatibilitási célokra használható. Ezeket a {$Z+} direktívával befolyásolhatjuk (azzal kérhetünk nagyobb ábrázolási módot).

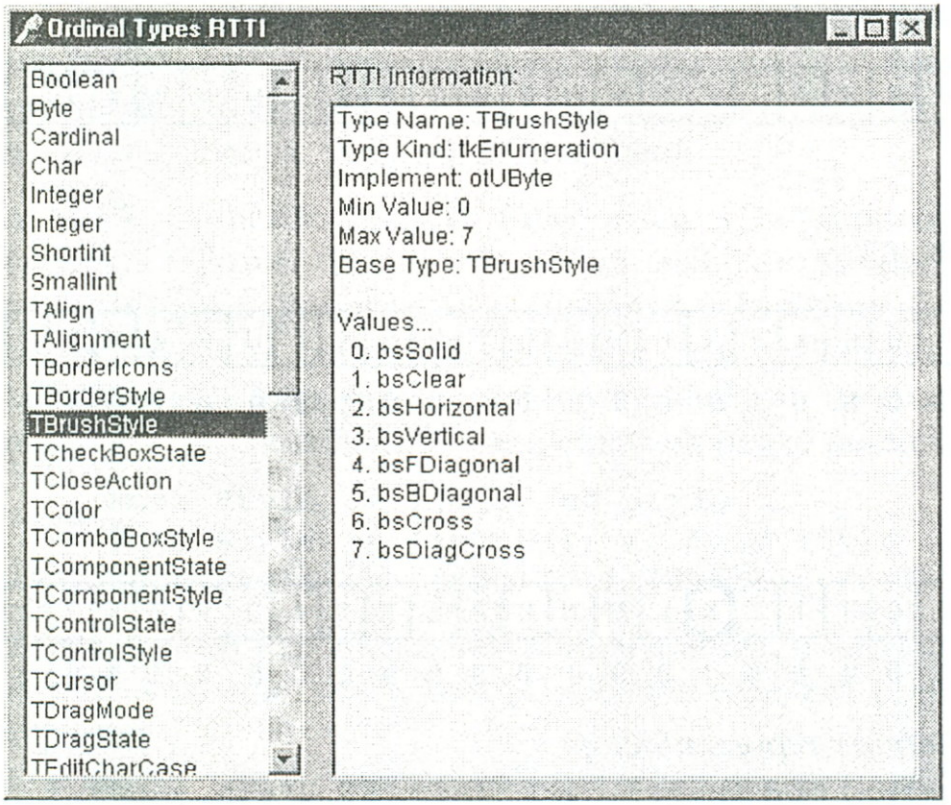
A Delphi VCL könyvtára felsorolás típusokat használ a legtöbb helyen. Például egy keret stílusa a következőképpen van meghatározva:

```
type
  TFormBorderStyle = (bsNone, bsSingle, bsSizeable,
    bsDialog, bsSizeToolWin, bsToolWindow);
```

Mikor egy tulajdonság értéke felsorolásszerűen van meghatározva, akkor az Object Inspector használatával választhatjuk ki a nekünk tetsző értéket.



A Delphi helpje általában kilistázza az elérhető sorszámozott típusokat és azok lehetséges értékeit. Egy másik módszerrel, a mellékelt CD TOOLS könyvtárában található OrdType programmal kilistázhatjuk az egyes Delphi felsorolt típusokat (enumeration, set, subrange) a hozzájuk tartozó értékekkel együtt. Az így kapható végeredményt a 4.8. ábrán láthatjuk.



4.8. ábra *A sorszámozott típusokkal kapcsolatos részletes információ, ahogy az a CD TOOLS könyvtárában található OrdType programmal látható*

Halmaz típusok

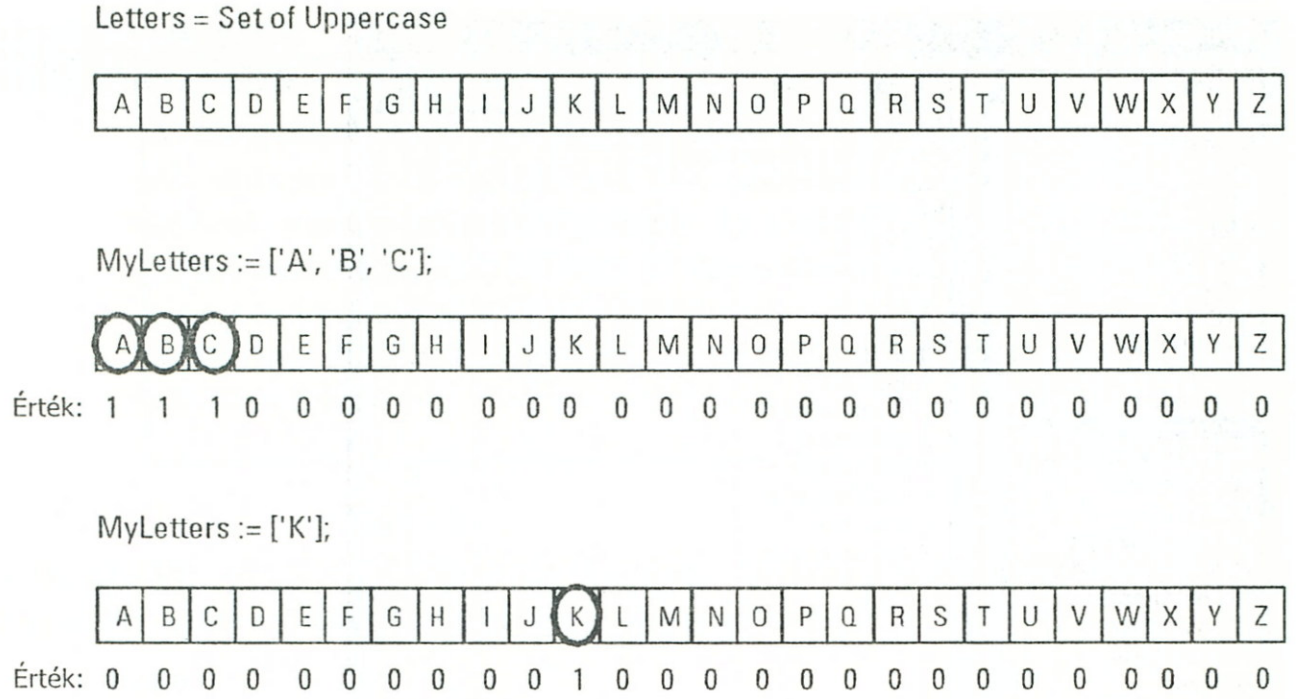
A halmaz típusok a sorszámozott, illetve gyakran a felsorolt és a részintervallum típusok értékeiből képzett típusokat jelölik, mivel nem tartalmazhatnak 256 darabnál több elemet. Egy változó általában egy, a típus intervallumába eső értéket tartalmaz. A halmaz ettől eltérően egyszerre nulla, egy, kettő, három, vagy ennél több értékét is tartalmazhatja egy adott intervallumnak, esetleg akár az egész intervallumot is magában foglalhatja. A halmaztípus fogalmának megértésére szolgál a 4.9. ábra és az alábbi kódrészlet:

```
type
  Letters = set of Uppercase;
```

Ezután már létrehozhatunk egy változót, amelyhez az eredeti típus több értékét is hozzárendelhetjük. A több érték hozzárendelése érdekében az értékeket szögletes zárójelben, vesszővel elválasztva kell megadni. Miután létrehoztuk a változót:

```
var
  MyLetters: Letters;
```

már hozzá is rendelhetjük a kívánt értékeket (jelen esetben először három, majd egy, és legvégül üres értéket rendelünk a változóhoz).

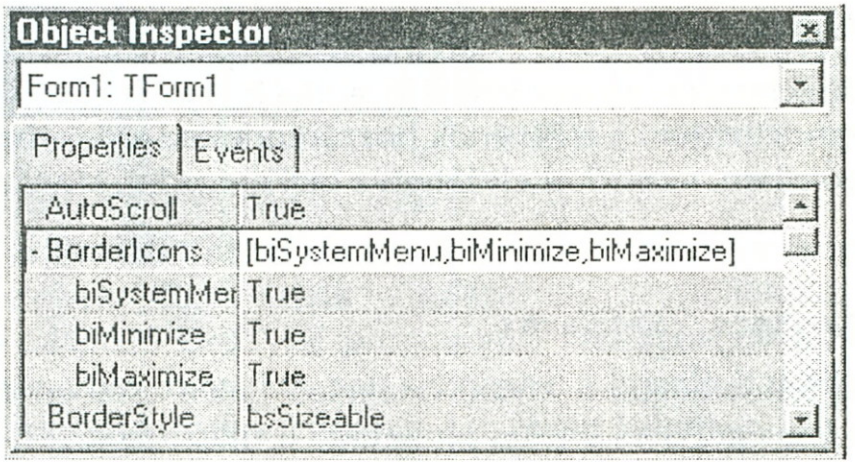


4.9. ábra *Egy halmaz reprezentációja*

```
MyLetters := ['A', 'B', 'C'];
MyLetters := ['K'];
MyLetters := [];
```

Delphiben a halmaz típust legtöbbször állapotinformációk tárolására használjuk. A következő kétsoros programrészlet (amely a Delphi rendszerkönyvtárának részét képezi) a kereten elhelyezhető ikonokat és egy, annak megfelelő halmazt határoz meg:

```
type
  TBorderIcon = (biSystemMenu, biMinimize, biMaximize,
      biHelp);
  TBorderIcons = set of TBorderIcon;
```



Természetesen, egy adott ablak az előbb felsorolt ikonok közül egyetlen egyet sem, egyet vagy többet is tartalmazhat egyszerre. Az Object Inspectort használva a megfelelő értékeken való dupla kattintással az adott értéket a csoporthoz hozzárendelhetjük, vagy abból kivehetjük.

A másik halmazon alapuló tulajdonság a betűtípusok stílusa. Lehetséges értékek a félkövér (bold), dőlt (italic), aláhúzott (understrike), és az áthúzott (strikethrough). Természetesen ugyanaz a betűtípus lehet dőlt és félkövér, tartalmazhatja az összes tulajdonságot vagy egyiket sem. Ebből kifolyólag a betűstílus is halmaz típusú tulajdonság által van meghatározva. Ehhez a halmazhoz a következőképpen rendelhetünk hozzá értékeket:

```
Font.Style := []; // no style
Font.Style := [fsBold]; // bold style only
Font.Style := [fsBold, fsItalic]; // two styles
```

A halmazokon más műveletek is végezhetőek, a hasonló halmaztípussal rendelkező változók összeadását is beleértve (egészen pontosan azok uniójának képzése):

```
Font.Style := OldStyle + [fsUnderline]; // two sets
```

Általában az összeadást fogjuk használni egy-egy halmaztípushoz való új érték hozzáadásához, ahogyan azt a következő, a 8. fejezetben található Choice példaprogramból kivett részleten láthatjuk:

```
procedure TForm1.CheckBoldClick(Sender: TObject);
begin
  if CheckBold.Checked then
    Memo1.Font.Style := Memo1.Font.Style + [fsBold]
  else
    Memo1.Font.Style := Memo1.Font.Style - [fsBold];
end;
```

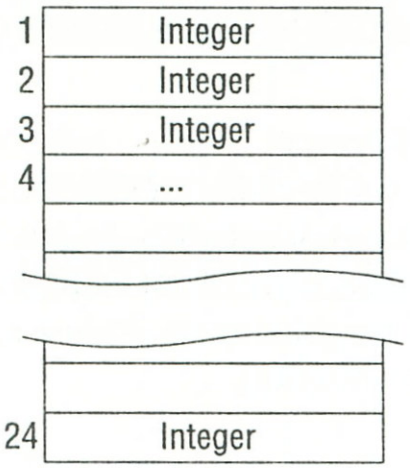
Tömb típusok (array types)

A tömb nem más, mint meghatározott számú, azonos típusú elemekből álló összetett adattípus (4.10. ábra). Például így hozhatunk létre egy 24 integerből álló tömböt:

```
type
  DayTemperatures = array [1..24] of Integer;
```

A tömb meghatározásához kapcsos zárójelben kell megadni a tömb alsó és felső indexhatárát. Mivel mind az alsó és a felső határt meg lehet adni, ezért nem feltétlenül szükséges mindig nullától kezdődő indexintervallumot használni, mint a C-ben, C++-ban vagy más programnyelvekben.

DayTemperatures = Array [1..24] of Integer;



4.10. ábra *Egy tömb reprezentációja*

Mivel az index részintervallummal van meghatározva, a Delphi ellenőrizheti az index helyes megadását, ahogyan azt a korábbi példákban már láthattuk. A rossz, részintervallumon kívüli index megadása esetén fordítás közben „Invalid constant subrange", vagyis „Érvénytelen konstans intervallum", futás közben pedig az „Out of range", vagyis „Intervallum túlcsordulás" hibaüzenetet kaphatjuk.

A fenti tömböt felhasználva a DayTemp1 változót a következőképpen állíthatjuk be:

```
DayTemp1 [1] := 54;
DayTemp1 [2] := 52;
...
DayTemp1 [24] := 66;
DayTemp1 [25] := 67; // compile-time error
```

Egy tömb több dimenziós is lehet, ahogyan azt alábbi példánk mutatja:

```
type
  MonthTemps = array [1..24, 1..31] of Integer;
  YearTemps = array [1..24, 1..31, Jan..Dec] of
                                          Integer;
```

Mindkét tömb ugyanazon a típuson alapul, ezért azokat megelőző adattípusok segítségével is létrehozhatjuk őket:

```
type
  MonthTemps = array [1..31] of DayTemperatures;
  YearTemps = array [Jan..Dec] of MonthTemps;
```

Ez a meghatározás megfordítja az indexek sorrendjét, és megengedi egész blokkok, változók közötti hozzárendelését is. Ezt mutatja be a következő kifejezés, amely a januári hőmérsékleti értékeket a februári tömbbe másolja.

```
var
  ThisYear: YearTemps;
begin
  ...
  ThisYear[Feb] := ThisYear[Jan];
```

Létrehozhatunk nulla indexhatárú tömböt is. Általában véve a több logikai határ használata előny, hiszen nem kell két elemet indexelni, hogy a harmadikhoz hozzáférjünk. A Windows, ezek ellenére, nulla indexhatárú tömböket használ (mivel a C programozási nyelven alapszik), és ezért nem meglepő, hogy a Delphi könyvtára hasonlóan működik.

Ha tömbökkel dolgozunk, úgy a tömb alsó és felső határát mindig meghatározhatjuk a Low és a High függvények segítségével. A Low és a High függvények használata ciklusok esetén kifejezetten ajánlott, hiszen az a tömb nagyságától független programrészlet írását teszi lehetővé. Később, ha megváltoztatjuk a tömb indexhatárait, a Low és a High függvényeknek köszönhetően a ciklus tökéletesen működik tovább. Ha azonban a ciklusba beleépítjük a tömb nagyságát, akkor a cikluson belüli értékeket állandóan módosítani kell, ha a tömb indexhatárait megváltoztatjuk.

Tipp

Mellékesen, a Low és a High értéke a fordítás közben számítódik ki: konstans érték képződik belőle, és nem egy függvényhívás, mint ahogyan azt sokan gondolnánk. Ez a fordítás közbeni átalakítás még egyes egyszerűbb rendszerhívások esetében is előfordulhat.

A Delphi tömböket főleg a különböző formokon használ. A TimeNow programban már láthattunk példát az ilyen tulajdonságokra, amelynél a ListBox elem Items tulajdonságait tömbön keresztül éri el. A Delphi ciklusainak áttekintésénél még foglalkozunk néhány további tömbtulajdonsággal is.

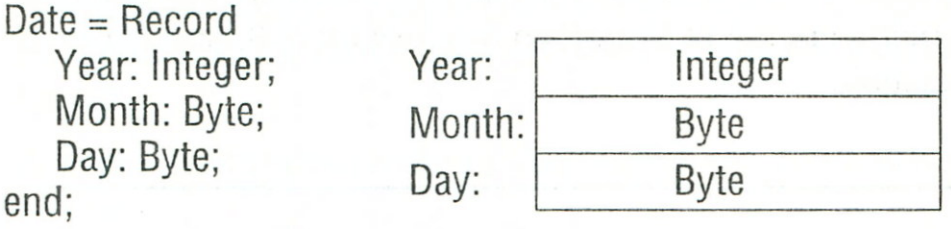
A rekord adattípus

A rekord tetszőleges számú és típusú komponensekből álló adatszerkezet, amelyben minden elemnek, vagy mezőnek megvan a saját típusa. A rekord deklarációja tartalmazza az összes komponens típusát és nevét, a későbbiekben ez utóbbi segítségével érhetőek el az egyes adatmezők.

Íme egy kisebb program egy rekord típus deklarációjával (4.11. ábra), egy, az adott típusú változó deklarálása, és néhány, a változót használó kifejezés:

```
type
  Date = record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;

var
  BirthDay: Date;
begin
  BirthDay.Year := 1997;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
```

Az osztályokat és objektumokat a rekord típus kiterjesztésének tekinthetjük. A Delphi könyvtárak általában osztályokat használnak a rekord adattípus helyett, de nagyon sok előre meghatározott rekord található a Windows API-jában is.



4.11. ábra *Egy rekord reprezentációja*

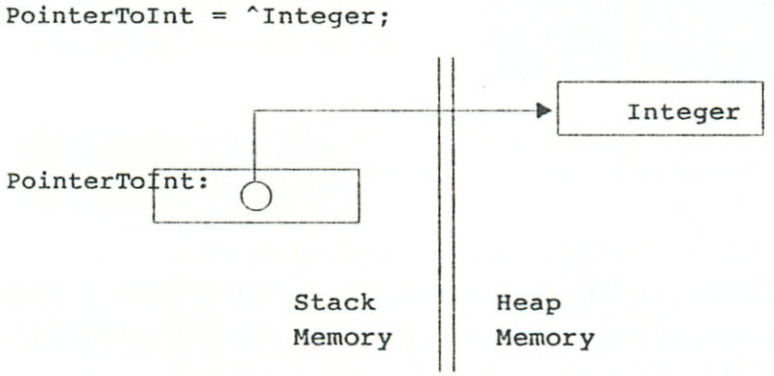
A rekord típusok ugyancsak rendelkezhetnek változó résszel, azaz, több mező mutathat ugyanarra a memóriarekeszre, még akkor is, ha különböző típussal rendelkeznek (ez a C nyelv union-jának felel meg). Alternatívaként, ezeket a mezőváltozatokat vagy mezőcsoportokat használhatjuk az egy rekordon belüli megegyező memóriaterület más szempontok alapján való eléréséhez is. Ennek a típusnak a fő felhasználási területe a kevésbben különböző adatok tárolása, és a típusmegfeleltetéshez hasonló hatás elérése. A változó rekordot manapság kiszorították az objektumorientált és egyéb programozási technikák, bár a Delphi egyes esetekben még mindig ezt használja.

A változó rekord típusok használata nem mindig teljesen biztonságos, ezért használatuk körültekintést igényel. Tapasztalt programozók azonban bátran alkalmazhatják ezeket, mert a Delphi könyvtárakon keresztül remekül használhatók. Ezektől függetlenül, inkább ne foglalkozzunk velük egészen addig, amíg nem érezzük magunkat gyakorlottnak.

A mutató típus (pointer)

A mutató típus olyan változót határoz meg, amely egy másik változó megadott, vagy meg nem határozott adattípusú memóriacímére mutat. Egyszóval a mutató típus közvetve ad meg egy értéket, ahogyan azt a 4.12 ábrán láthatjuk. A mutató deklarálásához az úgynevezett kalap (^) karaktert kell használnunk.

```
type
  PointerToInt = ^Integer;
```



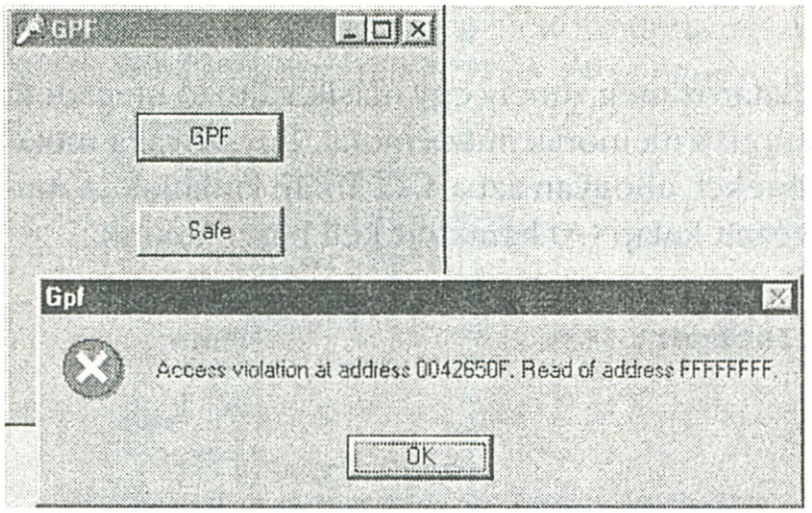
4.12. ábra

Egy mutató (pointer) reprezentációja

A mutató létrehozása után hozzárendelhetjük egy neki megfelelő típusú változó memóriacímét a @ operátor használatával, vagy a heap memórián belül új változót hozhatunk létre a New eljárás használatával. Ekkor azonban ne feledkezzünk meg a lefoglalt memória Dispose eljárással való felszabadításával, ha a változóra többé nincs szükségünk.

Ha egy mutatónak még nem adtunk meg értéket, akkor ahhoz hozzárendelhetjük a NIL (üres) értéket. Ezután mindig ellenőrizhetjük, hogy a mutató üres-e, vagy egy már létező értékre hivatkozik. Ez a módszer elég gyakran használatos, mivel egy nem létező vagy hibás mutatóra való hivatkozás eredménye legtöbbször hozzáférési hibát, avagy közismertebb nevén „General Protection Fault"-ot (általános védelmi hibát) eredményez. Ezt a hibát produkálja a kifejezetten erre a célra megírt GPF példaprogram (4.13. ábra), amely a következő programrészletet hajtja végre:

```
procedure TFormGPF.BtnGpfClick(Sender: TObject);
var
  P: ^Integer;
begin
  P := nil;
  ShowMessage (IntToStr (P^));
end;
```



4.13. ábra *A NIL pointerre történő hivatkozás esetén fellépő rendszerhiba*

Ugyanebben a programban láthatunk példát a biztonságos adatelérésre is. Itt a mutató egy külsőleg meghatározott változóra mutat és gond nélkül használható, de a biztonság kedvéért egy ellenőrzést is beiktattunk:

```
procedure TFormGPF.BtnSafeClick(Sender: TObject);
var
  P: ^Integer;
  X: Integer;
begin
  P := @X;
  X := 100;
  if P <> nil then
    ShowMessage (IntToStr (P^));
end;
```

Egy másik módszerrel a mutatóhoz dinamikusan is hozzárendelhetünk memóriacímet, mint ahogyan ezt a következő GPF példában tesszük:

```
procedure TFormGPF.BtnDynamicClick(Sender: TObject);
var
  P: ^Integer;
begin
  // initialization
  New (P);
  // operations
  P^ := 20;
  ShowMessage (IntToStr (P^));
  // termination
  Dispose (P);
end;
```

A Delphiben ugyancsak megtalálható a típus nélküli mutató típusa (mint a `void*` a C nyelvben). Ennek használata esetén a New eljárás helyett a GetMem eljárást kell használni. A GetMem eljárást mindig akkor kell meghívni, ha a memóriaváltozó nagysága nem meghatározott.

Az a tény, hogy a mutatók használata ritkán szükséges a Delphiben, a programozási környezet egy érdekes előnye. Ennek ellenére a mutatók működésének megértése és használatának elsajátítása elengedhetetlen szempont a professzionális programozási technikák megtanulásának és a Delphi objektummodelljeinek megértéséhez, amelyek a mutatókat a „színfalak mögött", a program mélyebb rétegeiben használják.

Megjegyzés

Bár általában nem sok pointert használunk a Delphiben, de sok hozzá hasonló struktúrát alkalmazunk, amelyet hivatkozásnak nevezünk. Az objektumok minden megjelenési formája közvetve egy pointer vagy hivatkozás, amely az objektum értékére mutat. Ennek ellenére ez teljesen el van rejtve a programozó elől, aki az objektumot ugyanúgy használja, mint bármely más adattípust. A Delphi objektum-hivatkozási modelljét a következő fejezetben vizsgáljuk behatóbban.

Állománytípusok

Egy másik, tipikusan Pascal típus a `file` típus. A fájlok a bemeneti/kimeneti eszközökkel kapcsolatosak (input/output), és a 29. fejezetben foglalkozunk velük bővebben. Röviden, egy új file adattípust a következőképpen adhatunk meg:

```
type
  IntFile = file of Integer
```

Ezután már megnyithatjuk az ehhez rendelt fizikai fájlt és értékeket olvashatunk belőle, vagy írhatunk bele. A fájlok használata a Pascalban eléggé szigorúan lett meghatározva, nem úgy, mint a Delphiben, amely néhány olyan elemet is tartalmaz, amelyek tartalmukat képesek fájlból betölteni vagy oda kimenteni. Ezek között vannak streameket, illetve adatbázisokat támogató eljárások és részek, de ezekkel csak a későbbi fejezetekben foglalkozunk.

Stringek a Delphiben

A stringek kezelése a Delphi nyelvben egészen egyszerűnek mondható, de a program működésének belső rétegén eléggé bonyolult. A Pascal a hagyományos módszerrel kezeli a stringeket, míg a Windows sajátságos módon, a C nyelvhez hasonlóan. A 32 bites Delphi mindkettőt támogatja, beleértve az alapértelmezett hosszú string (long string) adattípust is.

Először a hagyományos stringkezelést tárgyaljuk, majd áttérünk a Windows és a Delphi módszereire. A további szakaszokban ezeket a témaköröket fogjuk tárgyaljuk.

A hagyományos Pascal stringek

A Borland cég Turbo Pascaljában és a 16 bites Delphiben a tipikus string adattípus nem más, mint egy karaktersorozat, amely az első bájtban tartalmazza a string hosszát. Minden string fix méretű (alapbeállítás szerint ez 255 karakter), bár általában ennél kevesebbet használunk. A hagyományos Pascal stringek maximálisan 255 karaktert tárolhatnak, ami a legjelentősebb hátrányuk. Lássuk a következő példát:

```
var
  Name: string;
  Title: string [50];
```

A Name nevű változó egy 255 karakter hosszú string, míg a Title maximálisan 50 karakter befogadására képes. Ahogyan a szögletes zárójelek használatából is kitűnik, a string típus nagyon hasonlóan viselkedik a tömb típushoz. Tulajdonképpen a string nem más, mint egy karaktereket tartalmazó tömb. Ezért is hivatkozhatunk a string egyes elemeire (jelen esetben az első karakterre) a következőképpen:

```
FirstChar := Name [1];
```

Számos függvénnyel használhatunk, illetve módosíthatunk stringeket (a teljes listát a Delphi Súgó fájljai tartalmazzák). Könnyedén összeadhatunk két vagy több stringet a plusz jel felhasználásával:

```
FinalString := FirstString + ' ' + SecondString;
```

Ez utóbbi kifejezés két stringet egyesít, s közéjük egy üres karaktert szúr.

A Delphi hosszú stringjei

A hagyományos stringek hátrányainak kiküszöbölése érdekében a 32 bites Delphiben megjelenik a hosszú stringek használatának lehetősége. Ez három új string típust takar:

- A ShortString típus megfelel a hagyományos Pascal stringnek, amellyel az előbb foglalkoztunk. Maximális hosszúsága 255 lehet, és a 16 bites Delphi string típusának felel meg. A string minden egyes eleme ANSIChar típusú, vagyis az alap karaktertípus.
- Az ANSIString típus az új, változó hosszúságú string típus neve. Az ilyen típusú stringek számára a memória dinamikusan kerül kiosztásra, azaz méretük szinte korlátlan lehet (csak egy jó nagy memóriával rendelkező gépet kell vennünk). A string egyes elemei ugyancsak ANSIChar típusúak.
- A WideString típus hasonló az ANSIString típushoz, azzal a különbséggel, hogy itt a string elemei WideChar típusúak. Ez a stringtípus először a Delphi 3-ban jelent meg, és alkalmazásukra legtöbbször az OLE használatakor lesz szükség.

Megjegyzés

A VCL a legtöbb esetben támogatja a wide stringek használatát annak ellenére, hogy a Windows 95 operációs rendszernek elég gyengék a képességei e téren, és némely API függvény egyáltalán nem támogatja azokat. Ezzel szemben a Windows NT-n a wide stringek meglepően jól működnek! Hogy lássunk erre egy példát, futtassuk le a CD WideStr programját, amely a SetWindowTextW API függvényt, a SetWindowText API függvény wide stringes verzióját használja. Windows 95 alatt semmi hatása nem lesz a programnak, míg Windows NT alatt minden tökéletesen működik.

A Delphi 3-ban (hasonlóan a 2-höz) ha stringet használunk, akkor az automatikusan ShortString vagy ANSIString típusú lesz, a $H fordító direktíva értékétől függően. Az alapbeállítás szerinti $H+ ANSIString típusú stringek létrehozását eredményezi, amelyet a Delphi könyvtár elemei használnak.

Elméletileg, egy ANSIString típusú string maximum kétmilliárd karaktert tartalmazhat, többet, mint amennyit a 32 bites Windows egy-egy alkalmazásnak egyáltalán engedélyez. Ahogyan azt korábban említettük, ezeket a stringeket a memóriában dinamikusan kezeli a rendszer, így amikor egy stringet megduplázunk, akkor az új string helyett csak egy mutató jön létre, ami az eredeti stringre mutat. Amint azonban bármelyik stringben megváltoztatunk egy értéket, az egész string megduplázódik.

StrRef: Példa a dinamikus string allokálásra

A CD-n található StrRef című példa a stringek dinamikus allokálását (memória foglalását) mutatja be. A programban két globális stringet deklaráltam:

```
var
  FormStrRef: TFormStrRef;
  Str1, Str2: string;
```

Mikor az első gombot lenyomjuk, a program a konstans stringet az egyesnek, az egyest pedig a kettesnek adja értékül:

```
procedure TFormStrRef.BtnAssignClick(Sender:
TObject);
begin
  Str1 := 'Hello';
  Str2 := Str1;
  Label1.Caption := 'Str1: ' + Str1 +
    ' - ' + IntToStr (Integer (Str1));
  Label2.Caption := 'Str2: ' + Str2 +
    ' - ' + IntToStr (Integer (Str2));
  BtnChange.Enabled := True;
end;
```

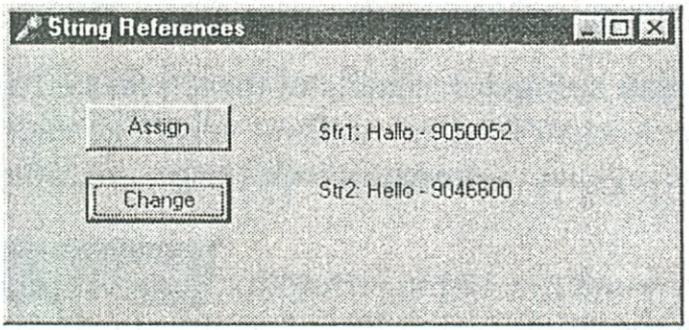
A többi kifejezés segítségével a két címke feliratának értékül adjuk a stringek tartalmát, azok értékének numerikus változatával együtt. Ezen értékek kiszámolásához egy kikényszerített típuskonverziót használtam, stringből egésszé. Mivel a stringek mindig mutatók, ezért az mindig egy memóriacímet tartalmaz.

A példa futtatása során két stringet kell kapnunk, megegyező tartalommal és memóriahellyel. Ha most megváltoztatjuk valamelyik string értékét, akkor a hozzá tartozó memóriahely is változni fog. Ezt a módszert copy-on-write technikának nevezik angolul.

Konkrétan úgy érhetjük el ezt a kimenetet, hogy a következő sorokat írjuk a második gomb OnClick eseménykezelőjébe:

```
procedure TFormStrRef.BtnChangeClick(Sender:
TObject);
begin
  Str1 [2] := 'a';
  Label1.Caption := 'Str1: ' + Str1 +
    ' - ' + IntToStr (Integer (Str1));
  Label2.Caption := 'Str2: ' + Str2 +
    ' - ' + IntToStr (Integer (Str2));
end;
```

Az utolsó két sor újra megjeleníti a két stringet, ezúttal az általunk kívánt formában.



Jegyezzük meg, hogy a BtnChangeClick metódust csak a BtnAssignClick metódus után hívhatjuk meg. Ennek betartatása végett a program írásának kezdetekor a második gomb nincs engedélyezve (azaz annak Enabled tulajdonsága hamisra van állítva), az első metódus engedélyezi futás közben.

A Delphi hosszú stringjei hivatkozásszámlálási mechanizmuson alapszanak, amely azt tartja számon, hogy egy-egy memóriában lévő stringre hány változó hivatkozik. Amennyiben ez a szám eléri a nullát – vagyis egyetlen egy változó sem hivatkozik rá –, úgy a string által lefoglalt memóriaterület felszabadíthatóvá válik.

A memóriában lévő string áthelyezésre kerül, ha a méretét megváltoztatjuk. Ez azt jeleni, hogy a teljes string átmásolódik egy másik helyre, ha az eredeti helyén már egyik irányban sem tud tovább terjeszkedni. Ezért megadhatjuk egy string maximális méretét a SetLength eljárással, azaz a string számára a megfelelő nagyságú memóriát foglalhatjuk le.

```
SetLength (String1, 200);
```

Megjegyzés

A SetLength eljárás memóriaigénylést hajt végre, de nem foglalja le aktuálisan a memória egy darabját, csak fenntartja azt a későbbi használatra. Ez a technika a Windows operációs rendszer által nyújtott lehetőségeken alapszik, és jellemző az összes Delphin belüli dinamikus memóriafoglalási módszerre. Például, amikor egy nagyon nagy tömböt hozunk létre, akkor annak memóriaterülete csak fenntartásra kerül, és nem lefoglalásra.

A string hosszának beállítása ritkán szükséges. Egyetlen esetben kötelező a hosszú stringek számára helyet foglalni a memóriában ilyen módon, ha az API függvénynek (a megfelelő típusátalakítás után) stringet kell paraméterként átadni. Ennek módszerét a későbbiek folyamán ismerjük majd meg.

A másik fontos dolog, amely a hosszú stringek használatát tünteti fel kedvező színben az, hogy a hosszú stringek egytől egyig nullával lezártak, ezért a Windows C nyelv nullával lezárt stringjeivel kompatíbilisek (lásd a következő szakaszt).

Ezen okok és a hosszú string mérete miatt a 32 bites Delphi a hosszú stringet használja általánosan, a hagyományos string csak a Pascal nyelvvel való kompatibilitási célok miatt maradt meg. Mivel az ANSIString az alapbeállítás szerinti, így nyugodtan használhatjuk a string adattípust karakterláncok létrehozásánál. Ettől függetlenül tartsuk észben, hogy a stringek milyen módon kerülnek beépítésre a programba, mert csak ennek ismeretében írhatunk korrekt és gyors programot.

C-stílusú karaktertömbök

Alternatívaként a string típusokhoz, használhatunk egy nulla alsó indexű karaktertömböt a nullával lezárt karaktersorozatok tárolására. A nullával lezárt string nem más, mint karakterek sorozata nulla értékű bájttal lezárva. A karaktertömb típusok általánosak a C nyelvben és kifejezetten gyakran használatosak a Windows API függvényekben.

Mivel a Pascal hosszú stringjei teljesen kompatibilisek a C nullával lezárt stringjeivel, a karaktertömbök használata a 16 bites alkalmazásokkal való kompatibilitás megtartására használható. Karaktertömböt a következőképpen hozhatunk létre:

```
var
  Name: array [0..50] of Char;
```

Ezután ezt már bármely ismert tömbkezelési eljárással módosíthatjuk. Mégis van egy második lehetőség a karakter alapú tömbök használatára. Ha az „extended language syntax" engedélyezve van (alapállapot szerint engedélyezett), akkor a nulla indexhatárú karaktertömb kompatíbilis a PChar típussal, ami egy karakterre hivatkozó mutató. Elcsodálkozhatunk azon, hogy ez miért is szükséges. Röviden azért, mert számos Windows API függvény PChar típusú paramétert feltételez a helyes működéshez, és a fentebb említett kompatibilitásnak köszönhetően létrehozhatunk egy karaktertömböt, feltölthetjük értékekkel, majd átadhatjuk az API függvénynek, mint paramétert.

Általánosságban, még mielőtt egy Windows API függvény számára egy mutatót adnánk át, memóriát kell foglalnunk a számára. A PChar stringek esetében a legegyszerűbb módszer az, hogy a mutató helyett egy karaktertömböt hozunk

létre, majd azt adjuk tovább az API függvénynek, vagy létrehozhatunk egy pointert, és dinamikusan foglalunk le számára memóriát a GetMem vagy a NewStr függvények segítségével. Ebben az esetben sohase feledkezzünk meg a memória felszabadításáról a Dispose eljárás segítségével, ha a pointert többé nem kívánjuk használni. Ezek ellenére a legegyszerűbb megoldás a Delphi hosszú stringek használata. Ne feledjük, hogy nagyon sok Delphi függvény nullával lezárt stringekkel dolgozik.

Figyelem!

Annak ellenére, hogy számos Windows függvénynek létezik egy PChar paramétere, egy PChar változó létrehozása és a függvénynek paraméterként való átadása egy eléggé súlyos hibát hoz létre. Ezzel a módszerrel ugyanis egy inicializálás nélküli mutatót adunk át a függvénynek, ami hibát okoz, amikor a Windows olvasni vagy írni akarja a stringet. Alapjában véve ez az, ami a „General Protection Failure" hibát produkálja a GPF példaprogramban.

String átalakítások

Ahogyan azt már korábban is említettük, a Delphi számos függvénye áll rendelkezésünkre a stringekkel való műveletvégzésre. Ezek teljes listája megtalálható a Delphi Súgójában, ezért nem is soroljuk fel őket teljes egészében ebben a könyvben. Összességében véve két nagy függvénycsoportról érdemes szót ejteni:

Az első csoportba a stringeket valamilyen szempontból kurtító, illetve vágó függvények tartoznak. Ezek az eredeti 16 bites Delphiben nem találhatók meg. A TrimLeft, TrimRight és Trim az üres szakaszokat vágják le a string bal, jobb, illetve mindkét oldaláról (a felsorolás sorrendjében, értelemszerűen).

A második csoportba az átalakításhoz használatos függvények tartoznak. Ezekre már láthattunk példát a megelőző oldalakon. Most csak két ilyen függvényt érdemes kiemelni: a StrPas, amely a nullával lezárt stringeket hagyományos Pascal formátumúvá alakítja, illetve a StrPCopy függvényt, amely az előbbivel ellentétes átalakítást végzi el. Ezt a két függvényt főleg a 16 bites Delphi alatt lehet kihasználni.

Amennyiben egy hosszú Pascal stringet szeretnénk PChar típusúvá konvertálni, úgy az egyszerű típusmegfeleltetés is elegendő. Az ellentétes irányhoz, a PChar és más karaktertömbök stringgé konvertálásához néha még ez sem szükséges. Egy form részének PChar stringbe másolását (a GetWindowText

API függvény használatával), majd az így kapott stringnek a gomb nevébe való másolását végzi el a következő programrészlet:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  S1: String;
begin
  SetLength (S1, 100);
  GetWindowText (Handle, PChar (S1), Length (S1));
  Button1.Caption := S1;
end;
```

Ezt a programot a mellékelt CD-n a LongStr név alatt találhatjuk meg. Azt azonban ne tévesszük szem elől, hogy ha nem sikerül memóriát lefoglalnunk a SetLength segítségével a string számára, akkor a program nagy valószínűséggel összeomlik.

Ha a PChar-t értékek átadására használjuk (nem értékek átvételére, mint az előbbi példában), akkor a forráskód még egyszerűbbé válik, ugyanis nem szükséges átmeneti stringet létrehozni és inicializálni. A következő egysoros program egy címke Caption tulajdonságát adja át egy API függvénynek egyszerű PChar megfeleltetéssel:

```
SetWindowText (Handle, PChar (Label1.Caption));
```

Megjegyzés

Amikor WideString típust Windows-kompatíbilis típussal kell megfeleltetnünk, akkor a PChar helyett a PWideChar típust kell használnunk az átalakításhoz. Az OLE és a COM programok létrehozásához ugyancsak hosszú stringeket kell használni.

Stringátalakítási nehézségek

A platformfüggőségük és a nagy méretük miatt nem ajánlatos az új stringek nagy tömegben való alkalmazása, kivéve, ha a Delphi 16 bites verziójával való teljes kompatibilitást akarjuk fenntartani. Ugyanakkor problémák léphetnek fel, ha egy hosszú stringet PChar típusúvá konvertálunk.

Lényegében arról van szó, hogy az átalakítás után mi felelünk a stringért és annak tartalmáért, s ehhez semmiféle segítséget nem nyújt a Delphi. Nézzük például a fentebbi programrészlet módosítását (Button1Click):

```
procedure TForm1.Button2Click(Sender: TObject);
var
  S1: String;
begin
  SetLength (S1, 100);
  GetWindowText (Handle, PChar (S1), Length (S1));
  S1 := S1 + ' is the title'; // this won't work
  Button1.Caption := S1;
end;
```

A program fordítás utáni futtatásakor kisebb meglepetésben lesz részünk: A gomb felirata az ablak eredeti fejlécét tartalmazza, a konstans stringben meghatározott szöveg nélkül. Ez onnan ered, hogy amikor a Windows egy stringbe ír, akkor nem állítja be megfelelően a hosszú Pascal string hosszúságát. A Delphi ettől függetlenül kitalálja, hogy merre található a string vége a nulla lezárásra alapozva, de ezért bármi mást fűzünk a stringhez, az a nulla lezárás mögé kerül, így azokat a Delphi figyelmen kívül hagyja.

Hogyan oldhatjuk meg ezt a problémát? Úgy, hogy arra utasítjuk a rendszert, hogy a GetWindowText API függvény által visszaadott stringet konvertálja Pascal kompatibilissé. De ha a következő programot írjuk:

```
S1 := String (S1);
```

akkor azt a rendszer figyelmen kívül hagyja, mert egy adattípust a saját maga típusába konvertálni értelmetlen és felesleges művelet. A megfelelő hosszúságú Pascal string előállításához először PCharrá kell azt átalakítanunk, majd hagyni, hogy a Delphi visszaalakítsa stringgé:

```
S1 := String (PChar (S1));
```

A string átalakítás akár el is hagyható, mert a PChar-string konverzió a Delphin belül automatikus. Íme a végleges forráskód:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  S1: String;
begin
  SetLength (S1, 100);
  GetWindowText (Handle, PChar (S1), Length (S1));
  S1 := String (PChar (S1));
  S1 := S1 + ' is the title';
  Button3.Caption := S1;
end;
```

Más módszerrel is beállíthatjuk a Delphi string hosszúságát a PChar hosszúságának alapján, a következő módon:

```
SetLength (S1, StrLen (PChar (S1)));
```

Mindhárom módszerre találunk példát a LongStr példaprogramban, amelynek minden egyes gombjához ezen módszerek egyikét rendeltük hozzá. Ha egy form nevéhez akarunk hozzáférni, akkor egyszerűen csak használjuk annak a Caption tulajdonságát, semmi szükség ilyen bonyolult és zavaró kódok megírására, amelyek a stringek átalakításainak bemutatására készültek. Néhány esetben azonban elengedhetetlenül szükséges a fentebb vázolt megoldást alkalmazni, főleg akkor, amikor Windows API függvényeket alkalmazunk.

Stringek formázása

A + jel és néhány átalakításra használható függvény (mint például az IntToStr) segítségével összetett szerkezetű stringek alakíthatók ki a már meglévő értékekből. Létezik azonban egy másik módszer a számok, pénznemek és más stringek végső formátumba való konvertálására. Ezekhez használhatjuk a Format függvényt, vagy annak társfüggvényeit.

A Format függvénynek paraméterként meg kell adni a formázni kívánt szöveget stringben, néhány pozíciójelölőt (általában a % jellel jelölik őket) és az értékeket tartalmazó halmazt. Például két szám egy stringgé formázásához a következőt kell írni:

```
Format ('First %d, Second %d', [n1, n2]);
```

ahol az n1 és n2 két Integer érték. A az első pozíciójelölő az első, a második pozíciójelölő a második értékkel helyettesítődik, és így tovább. Ha a pozíciójelölő típusa (a % utáni betűvel van meghatározva) nem egyezik meg a megfelelő paraméter típusával, úgy futásidejű hiba lép fel. Az ilyen hibák fordítás közbeni figyelmen kívül hagyása a Format függvény nagy hátránya.

Megjegyzés

A Format függvény nyitott-tömb paramétert használ (amelynek tetszőleges számú eleme lehet), amit ennek a fejezetnek a végén tárgyalunk bővebben. Most csak azt figyeljük meg, hogy a második paraméterként átadott értékének listája egy tömb felépítéséhez hasonló.

A %d-n kívül a 4.4. táblázatban felsorolt pozíciójelölők mindegyikét használhatjuk, tetszés szerint. Ezekkel egy adott külalak érhető el, de ennek további jellemzőit meg lehet változtatni. A szélesség meghatározó jellel például meg-

adhatjuk, hogy a kimenetben maximálisan milyen hosszú karakterlánc szerepeljen, amíg a precíziós jellel a tizedesjegyek száma adható meg. Például:

```
Format ('%8d', [n1]);
```

az n1 számot egy 8 karakter hosszú stringgé alakítja át, jobbra rendezéssel (használjuk a „-" jelet, ha balra szeretnénk rendezni a stringet).

4.4. táblázat

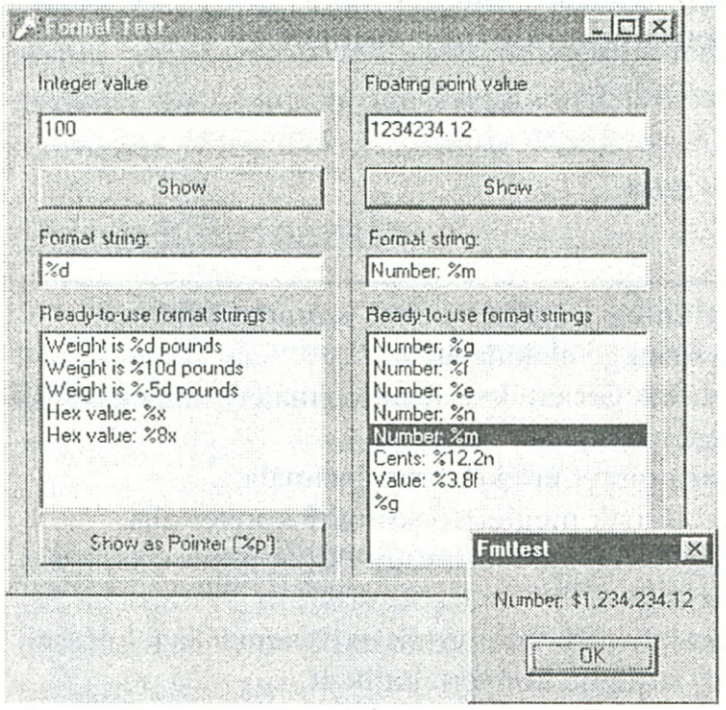
A Format függvény típusspecifikálói jelei

Jel	Jelentés
d (decimal)	a megadott Integer értéket a tízes számrendszernek megfelelő stringgé alakítja át.
x (hexadecimal)	A megadott integer értéket hexadecimális számjegyekből álló stringgé alakítja át.
p (pointer)	A megadott pointer értékét hexadecimális számábrázolásnak megfelelő stringgé konvertálja.
s (string)	A megadott string, karakter, vagy PChar átmásolódik a kimeneti stringbe.
e (exponential)	A megadott lebegőpontos érték exponenciális jelöléssel rendelkező stringbe konvertálódik át.
f (floating point)	A megadott lebegőpontos érték lebegőpontos jelöléssel rendelkező stringbe másolódik át.
g (general)	A megadott lebegőpontos értékből a lehető legrövidebb tízes számrendszerbeli számot jelképező szám képződik lebegőpontos, vagy exponenciális jelöléssel.
X (number)	A megadott lebegőpontos értékből lebegőpontos jelölésű string keletkezik, de ugyancsak jelölésre kerülnek az ezres pontok is.
m (money)	A megadott érték oly módon lesz átalakítva, mintha az pénznem volna. Az átalakítás a rendszer helyi beállításaitól függ. Ezzel kapcsolatban nézzük át a Delphi Súgó fájlját a „Currency and Date/Time formatting variables" címszó alatt.

A legegyszerűbben úgy láthatunk példákat ezekre az átalakításokra, ha eljátszogatunk velük. Ha erre esetleg nincs időnk vagy kedvünk, akkor keressük meg a CD-n a FmtText programot, amivel fix és lebegőpontos számok formázását végezhetjük el.

Ahogyan azt a 4.14. ábrán láthatjuk, ez a program egy két részre osztott formot hoz létre. A bal oldali részen az egész számok, a jobb oldali részen pedig a lebegőpontos számok helyezkednek el. Mindkét részen található egy szerkeszthető mező, ahol megadhatjuk, hogy milyen értéken akarunk formázást végezni, az alatta található gombbal pedig elvégezhetjük a formázást. A harmadik mezőben lehet megadni a formázás stringjét, vagy kiválasztani azt a legördülő

menüből. Ha itt egy új stringet adunk meg, akkor az a listához fog adódni (ne feledjük el, hogy a program bezárása után az újonnan hozzáírt értékek el fognak veszni). A harmadik mezőben láthatjuk a formázás utáni állapotot.



4.14. ábra *Egy példa az FmtFloat program lebegőpontos értékére*

A program teljes forráskódja túl bonyolult ahhoz, hogy itt elemzésre kerüljön, de ha már eléggé elmerültünk a Delphi lelkivilágában, akkor ennek megértése nem lehet nehéz feladat.

Programozási stílus

Mielőtt még tovább haladnánk a Pascal nyelvi elemek és kifejezések használatának rejtelmeiben, szükséges kiemelni a Pascal programozási stílusának néhány elemét. A kérdés, amelyre a választ meg fogjuk keresni az, hogy a megfelelő szintaktikai szabályok mellet hogyan kell rendes és rendszerezett forráskódot készíteni. Erre nem lehet egyszerűen válaszolni, hiszen a személyes tapasztalat, hozzáértés és ízlés nagyban befolyásolja a program külalakját. Ezek ellenére létezik pár olyan íratlan szabály, amit a programozók nagy része betart, hiszen az alkotói profizmusnak nem csak a program gyorsaságában, hanem annak forráskódjában és dokumentációjában is tükröződnie kell. Általában véve bármilyen programírási stílus végső célja a tisztaság és áttekinthetőség megteremtése kell, hogy legyen. Teljesen mindegy, hogy milyen stílusban is írjuk a programot, fontos, hogy annak a stílusnak a hatása érvényesüljön egész programunkban.

Megjegyzések

A Pascal nyelvben a megjegyzéseket kapcsos zárójelek, vagy a (*, *) pár közé kell beszúrni. A Delphi ugyancsak elfogadja a C++ stílusú megjegyzéseket, amelyeket a sor végére lehet csapni:

```
{ez egy megjegyzés}
(*ez is megjegyzés *)
// ez pedig sorvégi megjegyzés
```

A legelső a legtöbbet használt forma. A másodikat a legtöbbször az európai programozók használják, mert számos európai billentyűzetkiosztásnál nehezen érhető el a kapcsos zárójel. A harmadik a C++ nyelvből származik és csak a Delphi 32 bites verziójában használható. A harmadik típusú, a sorok végéig tartó megjegyzések alkalmasak rövid megjegyzések beszúrására, illetve egyes programsorok ideiglenes jelleggel való eltüntetésére.

Megjegyzés

Ebben a könyvben lehetőség szerint, a Delphi jelölésével egyetemben, a megjegyzések dőlt betűsek.

A három, egymástól különböző megjegyzési módszer segítségével egymásba ágyazott megjegyzések is készíthetőek. Például, ha figyelmen kívül akarunk hagyni pár programsort, amely tartalmaz megjegyzést, akkor nem használhatjuk ugyanazt a megjegyzés jelölést:

```
{  ... forráskód
{hiba a megjegyzésben}
... forráskód }
```

Egy másik fajta megjegyzés-jelöléssel már helyes programrészletet írhatunk:

```
{  ... forráskód
//ez már rendben van!
... forráskód }
```

Arra azonban vigyázzunk, ha a {-t vagy a (*-t dollár jel ($) követi, akkor az fordítási direktívát jelent, mint például a {$X+}.

A NAGY- és kisbetűk használata

A Pascal fordító (nem úgy, mint más nyelveké) nem figyeli a nagybetű-kisbetű különbségeket. Ezért a Myname, MyName, myname, myName és MYNAME azonosítók ugyanazt az objektumot jelentik. Egészében véve ez egy határozottan jó tulajdonság, hiszen más nyelvekben a legtöbb hibát a nem megfelelő nagy és kisbetű használata okozza.

Ennek ellenére néhány hátrányát is meg kell említeni a módszernek. Először is, hogy a fenti változók megegyeznek, ezért nem használhatóak különböző elemekre való hivatkozásokhoz. Másodszor, következetesen kell használni a kis és nagybetűket, hogy a forráskód olvasható legyen.

A nagybetűk következetes használatát a fordító nem követeli meg, de jó, ha használjuk. Az általában elfogadott szabály az, hogy csak az azonosító első betűjét kell nagybetűvel írni. Ha azonban az azonosító több szóból épül fel, úgy minden egyes szónak nagybetűvel kell kezdődnie:

```
MyLongIdentifier
MyVeryLongAndAlmostStupidIdentifier
```

Közök (white space)

A fordító által teljesen figyelmen kívül hagyott elemnek számítanak a szóköz, új sor és a tabulátor karakterek. Ezeket az elemeket összességében közöknek, esetleg fehér elválasztóknak nevezzük. Ezek nem befolyásolják a fordítás vagy a program futásának hatékonyságát, csak a forráskód olvashatóságát növelik nagymértékben.

A BASIC-kel ellentétben, a Pascal nyelvben egy kifejezést több sorba is írhatunk. Ennek hátránya (legalábbis a BASIC programozók szempontjából), hogy pontosvesszők használatával kell jelölni, hogy az egyes kifejezések meddig tartanak, vagy pontosabban, a kifejezéseket pontosvesszővel kell elválasztani egymástól. Az egyetlen megkötés ezzel kapcsolatban az, hogy string nem lehet egy sornál hosszabb.

Nincsen általánosan elfogadott, csak javasolt szabály a szóközök és több soron keresztül húzódó kifejezések használatára:

- A Delphi szerkesztőjében egy sor 60-70 karaktert tartalmaz. Lehetőleg mi se írjunk többet egy sorba, mert nyomtatáskor elég furcsa eredményeket kaphatunk.
- Ha egy eljárásnak vagy függvénynek több paramétere van, akkor a paramétereket egyenként külön-külön sorba írjuk.
- A megjegyzések előtt hagyjunk ki egy üres sort, vagy a nagyobb kódrészleteket bontsuk kisebb egységekre az üres sorok által. Ezzel

nagymértékben lehet növelni a forráskód olvashatóságát, mind a képernyőn, mind nyomtatásban.

Használjunk szóközöket a különböző paraméterek elválasztásához, vagy akár a zárójel megnyitása előtt. Lehet, hogy néhány programozó nem ért ezzel teljesen egyet, de erre csak azt lehet mondani, hogy a szóközök használata nem kerül semmibe, akkor pedig miért ne használjuk, ha vele olvashatóbb forráskódot hozhatunk létre? Igaz, hogy nagyobb helyet foglal el a merevlemezen, és tovább tart, ha a fájlt fel, illetve letöltjük modemen keresztül, de manapság ez már szinte teljesen elhanyagolható kérdés.

A szép külalak

A legutolsó tanács a fehér szóközökkel kapcsolatban a tipikus Pascal nyelven való programozás stílusához kapcsolódik. Más néven ezt szép külalaknak (pretty printing) nevezik. A szabály egyszerű: ahányszor egy logikailag összetett és beágyazott eljárást írunk, kezdjük azt 2 (vagy 4) karakterrel beljebb, mint az előtte lévő kifejezés:

```
if ... then
  statement;

if ... then
begin
  statement1;
  statement2;
end;

if ... then
begin
  if ... then
    statement1;
  statement2;
end;
```

Megjegyzés

A fenti formázás a szép külalakon alapul, de ezt a programozók saját stílusukkal is ötvözhetik. Néhányan a begin és az end utasításokat a belső programrészlettel egy vonalba írják, vannak, akik a hívó rutin szintjén, míg mások pedig a feltétel sorában helyezik el a begin szót. Ezek mind elfogadhatóak, és főleg a programozó egyéni ízlésétől függenek.

Hasonló módon kezeljük a változókat és egyéb adattípusokat is:

```
type
  Letters = set of Char;
var
  Name: string;
begin
    { long comment and long statement, going on in the
      following line and indented two spaces }
    MessageDlg ('This is a message',
      mtInformation, [mbOk], 0);
```

Természetesen, ezek az ajánlások csak a forráskód olvashatóságát növelik nagy mértékben, és semmiféle kihatással nincsenek a fordításra és a program végleges verziójára. A könyvben és a CD-n található összes példa ilyen szellemben és módszerrel íródott. A Delphi forráskódok, kezelési utasítások, és a Help is hasonló formákat követnek.

A szintaktikai elemek kiemelése

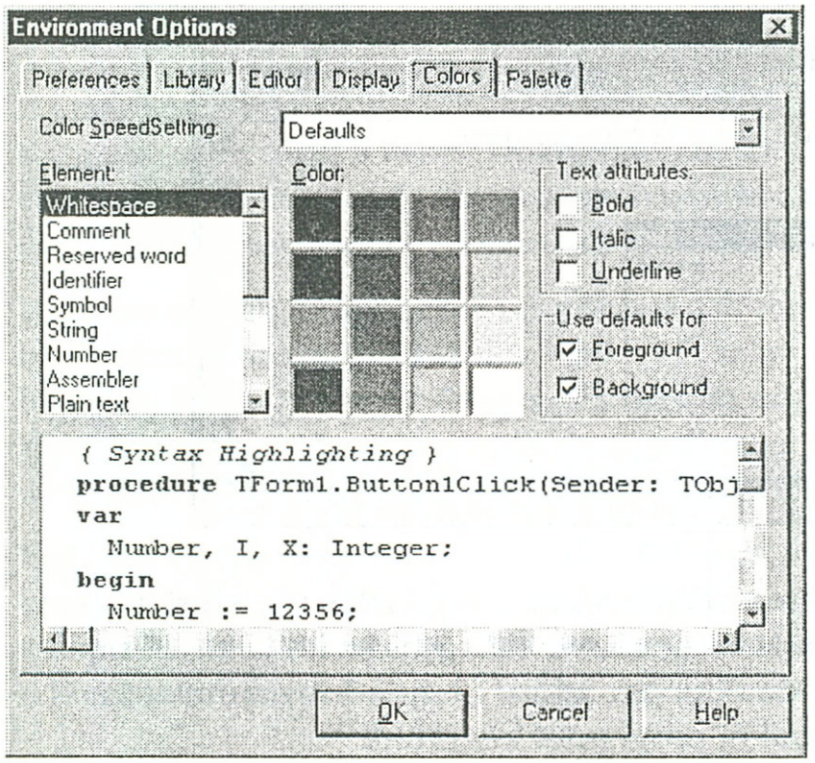
Hogy még könnyebben lehessen olvasni a Pascal forráskódot, a Delphi szerkesztőjében kihasználhatunk egy speciális lehetőséget. Ennek a neve szintaktikai elemek színes kiemelése (syntax color highlighting). Ez nem jelent mást, mint az egyes szavak jelentésüktől függően más és más színnel jelennek meg a képernyőn. Alapbeállítás szerint a kulcsszavak félkövérek, a stringek és megjegyzések más és más színnel jelöltek és gyakran dőlt betűsek, és így tovább.

Ennek használatával elég könnyen ellenőrizhetjük egy adott kulcsszó helyességét, a stringek helyes lezárását és a több soron keresztülhúzódó megjegyzések hosszát.

A színeket saját ízlésünk szerint változtathatjuk meg az Environment Options párbeszédablak **Editor Colors** oldalán (4.15. ábra) keresztül. Ha csak egyedül használjuk a programot, akkor olyan színeket úgy állítjuk be, amilyet akarunk. Ha más is használja rajtunk kívül, akkor lehetőleg egyeztessünk a többi programozóval egy egységes színkijelölés felállításáról, mert aki már dolgozott ilyen fajta rendszerrel, az tudja, hogy a megszokottól eltérő színkijelölés mellett dolgozni meglehetősen nehéz.

Megjegyzés

A könyv példaprogramjaiban kiemeléseket is alkalmaztunk, hogy a forráskódok sokkal olvashatóbbak legyenek. Reméljük, ezt sikerült is elérnünk.



4.15. ábra

A párbeszédablak, ahol a színes kiemelés jellemzőit állíthatjuk be

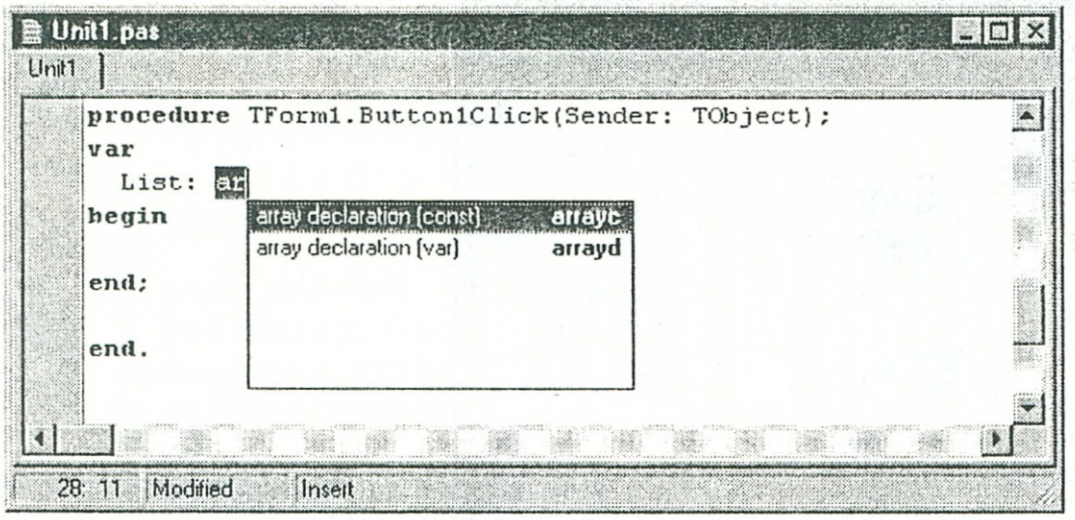
Kódsablonok használata

A Delphi 3 egy új tulajdonsággal rendelkezik a forrásprogramok szerkesztése terén. Mivel programírás közben gyakran megismételhetjük ugyanazt a kulcsszó sorozatot, a Borland egy új funkcióval, a Code Templates-cel látta el termékét, ahogy azt a 2. fejezetben már láttuk. Egy kódsablon nem más, mint egy rövidítés; a sablonhoz tartozó sorokat úgy jeleníthetjük meg, hogy beírjuk a szerkesztőben a rövidítést, majd megnyomjuk a CTRL+J billentyűket.

Például írjuk be, hogy `arrayd`, majd nyomjuk meg a CTRL+J kombinációt. Erre a szerkesztő a következő sort jeleníti meg az általunk beírt helyén:

```
array [0..] of ;
```

Mivel a kódsablonok sokszor ugyanannak a struktúrának a különböző változatait tartalmazzák, a rövidítés végén található betű általában arra utal, hogy melyik verziót szeretnénk használni. Azt is megtehetjük, hogy csak a rövidítés elejét gépeljük be (például annyit, hogy `ar`), és így nyomjuk meg a CTRL+J-t. Ekkor a szerkesztő egy menüt fog megjeleníteni, amelyből kiválaszthatjuk a nekünk kellő verziót, ahogy azt az ábrán is láthatjuk:



A sablonok teljes mértékben átalakíthatóak a saját ízlésünk szerint a nekünk tetsző elemek hozzáadásával, illetve kihagyásával. Ha módosítjuk, ne felejtsük el, hogy a '|' karakter azt jelöli, hogy a kurzor hova kerül, amikor a sablon használatával új programot készítünk.

Nyelvi kifejezések

Miután létrehoztunk néhány változót, azokat használhatjuk a programon belüli kifejezésekben. A Pascal lehetőséget biztosít számos állítás és kifejezés használatára. Először nézzük át a kifejezéseket és operátorokat.

Kifejezések és operátorok

Nem létezik általános szabály a kifejezések felépítésére, hiszen az attól függ, hogy milyen operátorokat használunk, és a Pascalban számos operátor található. Ezek az aritmetikai (matematikai), Boolean (logikai), reláció és halmaz operátorok, és még néhány másik. A kifejezések egyes változók értékadásához, eljárás vagy függvény paramétereinek kiszámolásához, vagy állapotok kiértékeléséhez használhatók, és ezeken kívül tartalmazhatnak állapot meghatározásokat is. Ahányszor egy azonosítón műveletet végzünk, és nem egymagában használjuk, úgy azt kifejezésnek tekinthetjük.

A kifejezések általánosan használtak a legtöbb programozási nyelvben. Kifejezésnek vehető egy tetszőleges konstans, változó, szöveges érték, operátor vagy függvény visszatérési értékéből képzett kombináció. Kifejezést legtöbbször meg lehet adni függvény vagy eljárás paramétereként, de nem minden esetben (ezek közé tartozik az is, mikor egy konkrét értéket kell megadni viszonyítási alapul).

Operátorok és a precedencia szabály

Ha már legalább egy programot írtunk életünkben, tudjuk azt, hogy mi is az a kifejezés, ezért most a Pascal operátorainak specifikus elemeit fogjuk részletesebben áttekinteni. A 4.5. ábrán a nyelvben használható operátorokat a precedencia szabályok szerint csoportosítva láthatjuk.

Vigyázzunk arra, hogy egyes operátoroknak más és más jelentése lehet különböző adattípusoknál. Például, a + jelet két szám összeadására, két string összefűzésére, halmazok egyesítésére, és még PChar pointerhez eltolási érték hozzáadására is lehet használni. Mindezek ellenére nem lehet két karaktert összeadni, mint ahogyan az lehetséges a C nyelvben.

Egy másik különleges operátor a div. A Pascal nyelvben tetszőleges két számot eloszthatunk egymással a / operátor használatával, úgy, hogy mindig valós számot kapunk vissza. Ha a két szám egymással való elosztása után egész számot szeretnénk eredményül kapni, úgy inkább használjuk a div operátort.

4.5. táblázat

A Pascal nyelv operátorai, elsőbbségi szabály szerint csoportosítva

Operátor	Felhasználási terület
Unáris (egyoperandusú) operátorok (legmagasabb prioritás)	
@	A változó vagy függvény címét adja vissza. (pointerrel tér vissza)
not	Boolean vagy logikai nem
Multiplikatív és bitenkénti operátorok	
*	szorzás, vagy halmazok esetében metszetképzés
/	Lebegőpontos osztás
div	Egészosztás
mod	Modulus (az egészosztás utáni maradékot adja vissza)
as	Típus-biztos típusmegfeleltetés (RTTI) operátor
and	Logikai vagy bitenkénti ÉS
shl	bitenkénti balra léptetés
shr	bitenkénti jobbra léptetés
Additív operátorok	
+	Aritmetikai összeadás, metszetek uniójának képzése, stringek összefűzése, pozitív előjel, illetve pointer pozitív irányban való eltolása
-	Aritmetikai kivonás, halmaz különbségképzése, negatív előjel, pointer eltolás kivonása
or	Logikai, vagy bitenkénti VAGY
xor	Logikai, vagy bitenkénti KIZÁRÓ VAGY

4.5. táblázat

A Pascal nyelv operátorai, elsőbbségi szabály szerint csoportosítva (folytatás)

Relációs és összehasonlító operátorok (legalacsonyabb prioritás)

Operátor	Felhasználási terület
=	Egyenlőség
<>	Nem egyenlő
<	Kisebb mint
>	Nagyobb mint
<=	Kisebb mint vagy egyenlő, illetve szűkebb részhalmaza
>=	Nagyobb mint vagy egyenlő, illetve bővebb részhalmaza
in	Igaz értékkel tér vissza, ha a megadott elem a halmaz eleme
is	Igaz értékkel tér vissza, ha az objektum típus-kompatibilis (szintén RTTI operátor)

Halmaz operátorok

A halmaz operátorok közé tartozik az unió (+), különbség (-), metszet (*), tartalmazás (in) és néhány összehasonlító operátor. Egy elem halmazhoz való hozzáadásához vehetjük a halmaz unióját egy olyan halmazzal, amely csak a kívánt elemet tartalmazza. Íme egy fontokhoz kapcsolódó Delphi példa:

```
Style := Style + [fsBold];
Style := Style + [fsBold, fsItalic] - [fsUnderline];
```

Ugyanezt megtehetjük a standard Include és Exclude eljárásokkal is, amelyek sokkal hatásosabban működnek, de nem alkalmazhatók részhalmaz típusú elemtulajdonságokkal, mert egyértékű paramétert várnak:

```
Include (Style, fsBold);
```

Egyszerű és összetett kifejezések

Egy Pascal kifejezés akkor egyszerű, ha az nem tartalmaz más kifejezést. Ilyenek például az értékadások és az eljáráshívások. Az egyszerű kifejezéseket egymástól pontosvessző választja el.

```
X := Y + Z;
Randomize;
```

Ahogyan azt már említettük az első fejezetben, a Pascal az értékadás kifejezésére kettőspont-egyenlőségjel kombinációt használ, egy furcsa jelölést a más programozási nyelvekhez szokott programozók számára. Ennek az az oka, hogy a sima egyenlőségjelet, amely más nyelvekben értékadást jelöl, a Pascalban az egyenlőség vizsgálatára használjuk.

Megjegyzés

Értékadásnál és kiértékelésnél a különböző jelek használatával a C fordítóhoz hasonlóan a Pascal fordító is gyorsabban végez a forráskóddal, ugyanis ebben az esetben nem kell megvizsgálni, hogy mit tartalmaz az a rész, amelyen az operátort használtuk. A különböző operátorok használatával ráadásul még tetszetősebb, jobban olvasható forráskódot kapunk.

Általában véve, a kifejezések egy összetett kifejezés részei, elejükön és végükön zárójellel lezárva. Az összetett kifejezés az általános Pascal kifejezés helyén is megjelenhet. Íme egy példa:

```
begin
  A := B;
  C := A * 2;
end;
```

Az utolsó kifejezés után található pontosvessző kirakása nem feltétlenül szükséges:

```
begin
  A := B;
  C := A * 2
end;
```

Mindkét verzió szabályos. Az egyetlen különbség az, hogy a fentebbi példa tartalmaz egy haszontalan (de veszélytelen) pontosvesszőt. A pontosvessző, gyakorlatilag nem más, mint egy kód nélküli kifejezés. Megjegyzem, hogy időnként ezek az üres kifejezések cikluson belül, és más esetekben is remekül használhatók.

Tipp

Bár ezeknek a pontosvesszőknek semmi szerepük sincsen, általában nem baj, ha használjuk őket, ugyanis néha előfordulhat, hogy írunk néhány sort és ahhoz egy vagy több utasítást akarunk hozzáadni. Ha a legutolsó pontosvessző hiányzik, akkor azt úgyis hozzá kell adni, ezért miért is ne rakjuk ki akkor, amikor az eredeti részt létrehozzuk?

Elágazások

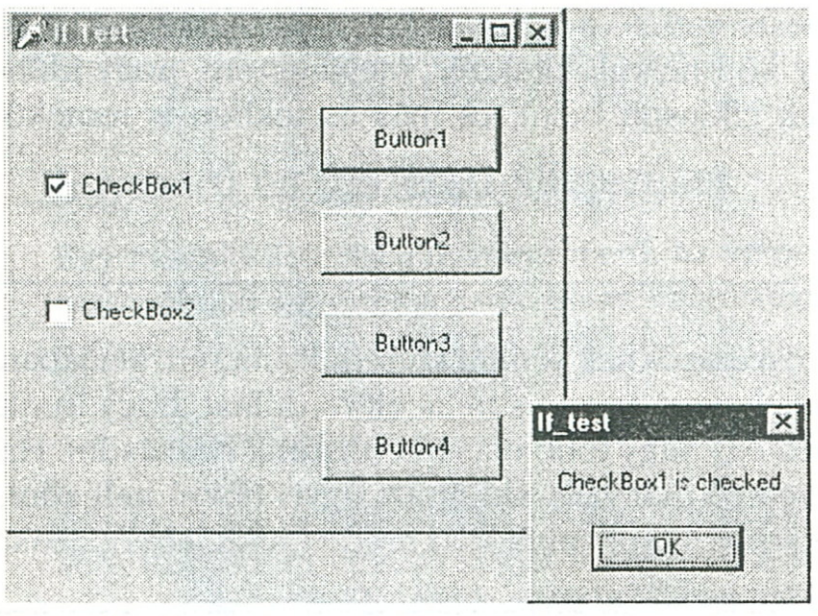
Az elágazásokkal bizonyos kifejezéseket hajthatunk végre, bizonyos feltételektől függően. Két fő feltételes utasítást kell megemlíteni: az if és a case utasításokat.

Az if utasítás

Az if utasítással bizonyos feltételek teljesülése esetén más kifejezéseket hajthatunk végre (if-then), illetve választhatunk két különböző kifejezés közül (if-then-else), ahol a feltételeket logikai kifejezésekkel adjuk meg. Egy egyszerű Delphi példán keresztül megtanulhatjuk a feltételes kifejezés írását. Először egy teljesen üres alkalmazást hozunk létre, majd két rádió-, és négy szokásos gombot helyezünk el rajta. Lehetőleg ne változtassuk meg a gombok neveit, de duplán kattintsunk mindegyiken, hogy létrehozzuk az OnClick esemény eseménykezelőjét. Íme az első gombhoz kapcsolódó egyszerű if utasítás:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  {simple if statement}
  if CheckBox1.Checked then
    ShowMessage ('CheckBox1 is checked')
end;
```

Amikor egy gombra kattintunk és az első rádiógomb aktív, akkor a program egy egyszerű üzenetet jelenít meg (4.16. ábra). A példában a ShowMessage függvény szolgál erre a célra, mert ez a legegyszerűbb Delphi függvény, amivel üzenetet lehet kiíratni a képernyőre.



4.16. ábra *Az IfTest példaprogramban ez az üzenet jelenik meg, ha az első rádiógomb aktív, és az első gombra kattintunk*

Ha a gombra katintunk és semmi sem történik, az azt jelenti, hogy a választómezőt nem kapcsoltuk be. Általában véve, nem árt, ha erre az eshetőségre felkészülünk a programban, mint ahogyan azt a második gomb esetében tesszük, amely már az if-then-else kifejezést használja:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  {if-then-else statement}
  if CheckBox2.Checked then
    ShowMessage ('CheckBox2 is checked')
  else
    ShowMessage ('CheckBox2 is NOT checked');
end;
```

Vigyázzunk arra, hogy nem tehetünk pontosvesszőt az első kifejezés és az else kulcsszó közé, mert abban az esetben a fordító hibaüzenettel megáll. Az if-then-else utasítás egyetlen egyszerű kifejezés, ezért nem helyezhetünk el benne pontosvesszőt.

Az if utasítás néha egészen bonyolult lehet. A feltétel akár több feltételből és kifejezésből is állhat, illetve az if utasítás egy másik if utasítást hívhat meg és így tovább. A legutolsó két gomb ezeket az eseteket testesíti meg:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  {statement with a double condition}
  if CheckBox1.Checked and CheckBox2.Checked then
    ShowMessage ('Both check boxes are checked')
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  {compound if statement}
  if CheckBox1.Checked then
    if CheckBox2.Checked then
      ShowMessage ('CheckBox1 and 2 are checked')
    else
      ShowMessage ('Only CheckBox1 is checked')
  else
    ShowMessage (
      'Checkbox1 is not checked, who cares for
                                  Checkbox2?')
end;
```

Nézzük át figyelmesen a program felépítését, majd futtassuk le. Ha a legkisebb kétségünk is felmerül ezek használatával kapcsolatban, úgy egy ehhez hasonló egyszerű program megírása sokat segíthet a dolog megértésében. Ha az alapprogram működik, akkor annak bonyolultságát saját elképzeléseink szerint növelhetjük.

A Case utasítás

Ha az if utasításunk túl bonyolult, akkor azt helyettesíthetjük a case utasítással. A case utasítás egy kifejezést, és az általa felvehető értékek egy részét vagy összességét tartalmazza. Ezek az értékek konstansok, és egyedi, megszámlálható típusúaknak kell lenniük. Használhatunk else kifejezést is, ami akkor hajtódik végre, ha a egyetlen érték sem felel meg az eredeti kifejezéssel. Két példa a case utasítás használatára:

```
case Number of
  1: Text := 'One';
  2: Text := 'Two';
  3: Text := 'Three';
end;

case MyChar of
  '+' :      Text := 'Plus sign';
  '-' :      Text := 'Minus sign';
  '*', '/': Text := 'Multiplication or division';
  '0'..'9': Text := 'Number';
  'a'..'z': Text := 'Lowercase character';
  'A'..'Z': Text := 'Uppercase character';
else
  Text := 'Unknown character';
end;
```

Ciklusok a Pascalban

A Pascal programozási nyelvben is megtalálhatóak azok az utasítások, melyekkel ugyanazon programrészlet többszörös, ismételt végrehajtása érhető el. Ezek a for, while és a repeat utasítások. Ezek használatával a legtöbb, más nyelveket ismerő programozó tisztában van, így csak egy rövid kitérő teszünk, hogy átfussuk legfontosabb jellemzőiket.

A for ciklus

A for ciklus a Pascalban egy számlálón alapszik, melynek értéke minden egyes lépésben nő vagy csökken. A következő egyszerű program az első tíz számot adja össze:

```
K := 0;
for I := 1 to 10 do
  K := K + I;
```

Ugyanez, csak a számláló visszafelé számoltatásával:

```
K := 0;
for I := 10 downto 1 do
  K := K + I;
```

A Pascal ciklusa nem annyira rugalmas, mint más nyelveké (nem lehetséges egytől különböző számmal csökkenteni, illetve növelni a számlálót), de egyszerűbben megtanulható és kezelhető. Ha egy sokkal bonyolultabb feltételek alapján működő ciklust szeretnénk létrehozni, akkor válasszuk a while vagy a repeat utasítást a for helyett.

A while és a repeat utasítások

A while-do és a repeat utasítások közötti alapvető különbség abban rejlik, hogy a repeat utasítással felépített ciklus minden esetben legalább egyszer lefut. A következő példa alapján ezt könnyen megérthetjük:

```
while I < 100 and J < 100 do
begin
  // use I and J to compute something...
  I := I + 1;
  J := J + 1;
end;
repeat
  // use I and J to compute something...
  I := I + 1;
  J := J + 1;
until I > 100 or J > 100;
```

Ha az I és a J eredeti értéke nagyobb mint 100, akkor a repeat ciklusban található utasítások egyszer mindenképpen végrehajtódnak.

Figyelem!

A másik jelentős különbség a kétfajta ciklus között, hogy a repeat-until ciklus ellentétes feltétellel rendelkezik, ugyanis a ciklus addig hajtódik végre, amíg a kifejezés igaz nem lesz. Ezzel szemben a while-do ciklus egészen addig fut, amíg a kifejezés igaz. Ebből kifolyólag a fenti példában meg kellett változtatni a kifejezést ugyanannak a célnak az elérése érdekében.

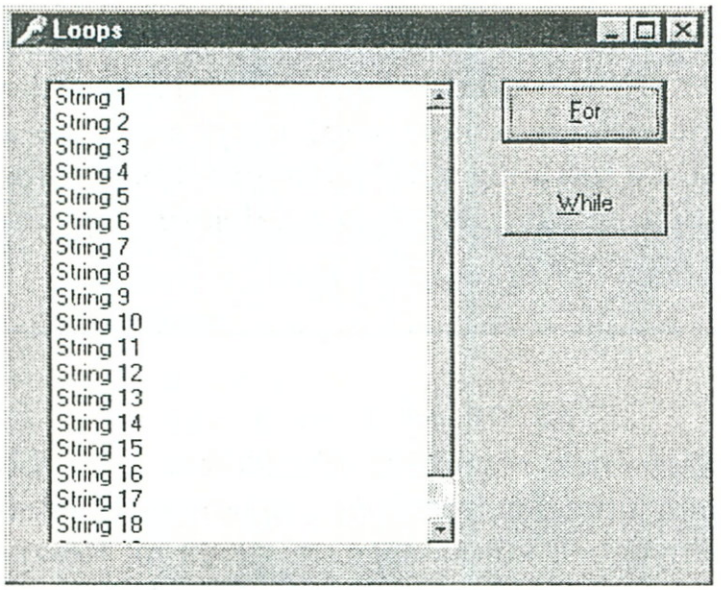
Hogy megértsük a ciklusok működésének részleteit, nézzük ezt a kisebb Delphi példát. A Loops példaprogram a meghatározott és a majdnem véletlen számlálót használó ciklus közötti lényegbeli különbségekre világít rá. Készítsünk egy teljesen új és üres alkalmazást, majd a formon helyezzünk el egy listát és két gombot. A gomboknak adjuk neveket (BtnFor és BtnWhile), majd

töröljük ki a Btn szót a Caption tulajdonságból, majd adjuk hozzá az & jelet, hogy aktiváljuk a billentyűkombinációt. A form szöveges leírása a következő:

```
object Form1: TForm1
  Caption = 'Loops'
  object ListBox1: TListBox ...
  object BtnFor: TButton
    Caption = '&For'
    OnClick = BtnForClick
  end
  object BtnWhile: TButton
    Caption = '&While'
    OnClick = BtnWhileClick
  end
end
```

Ezután már hozzáadhatjuk a gombokhoz a megfelelő eseménykezelő programrészleteket. Az első gombhoz egy egyszerű for utasítást rendelünk, amely nemes egyszerűséggel számokat listáz ki (4.17. ábra). Mielőtt még ezt a listához néhány számot hozzáfűző ciklust végrehajtanánk, ki kell törölnünk a lista teljes előző tartalmát:

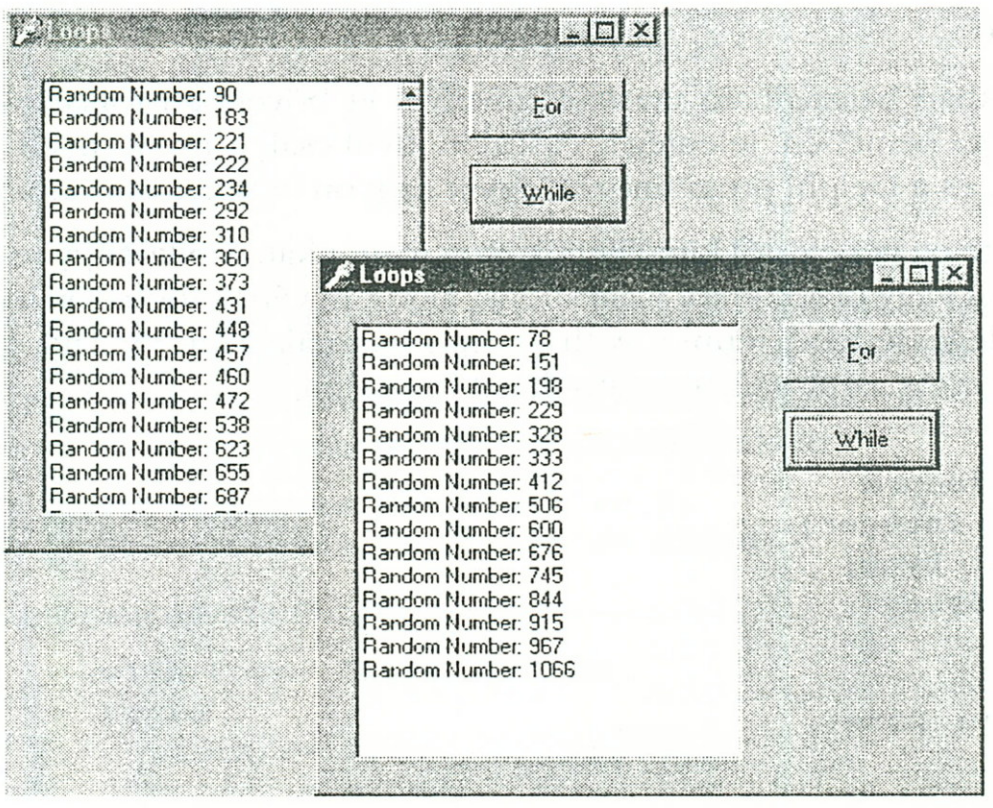
```
procedure TForm1.BtnForClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  for I := 1 to 20 do
    Listbox1.Items.Add ('String ' + IntToStr (I));
end;
```



4.17. ábra *Minden egyes esetben, ha a Loops példaprogram For gombjára kattintunk, az ablakban egymás után következő számok jelennek meg*

A második gombhoz rendelt utasítássorozat egy kicsit bonyolultabb, ugyanis a ciklus számlálója véletlen értékekkel növekszik minden egyes körben. Ennek eléréséhez először a Randomize eljárást kell meghívni, ami újraindítja a véletlenszámgenerátort, majd a Random függvényt 100-as paraméterrel. Ennek hatására egy 0 és 99 közötti számot kapunk, teljesen véletlenszerűen. A véletlenszámok sorozata (és főleg összege) befolyásolja, hogy a ciklus hányszor hajtódik végre.

```
procedure TForm1.BtnWhileClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  Randomize;
  I := 0;
  while I < 1000 do
  begin
    I := I + Random (100);
    Listbox1.Items.Add ('Random Number: ' + IntToStr
        (I));
  end;
end;
```



4.18. ábra *A Loops példaprogram listájában a While gomb minden egyes kattintásával egymástól különböző számok jelennek meg. A véletlenszám-generátor használata miatt ezek mennyisége és értéke különböző minden egyes esetben más*

Minden egyes esetben, ha a While gombra kattintunk, a számok különbözőek lesznek, mert a véletlenszám-generátortól függenek. A 4.18 ábrán láthatunk erre példát. Azt is megfigyelhetjük, hogy nem csak a számok különböznek minden egyes esetben, hanem azok mennyisége is. Ez azért van, mert a ciklus nem mindig ugyanannyiszor fut le, a véletlen számoknak köszönhetően. Ezért, ha egymás után többször is megnyomjuk a While gombot, akkor esetenként több, vagy kevesebb sort láthatunk.

Megjegyzés

A ciklus futását mindig befolyásolhatjuk a Break és Continue eljárások használatával. Az előbbi megszakítja a ciklust, míg a második eggyel növeli a ciklusváltozó értékét és közvetlenül a ciklus elejére helyezi a végrehajtás menetét (kivéve ha a ciklusváltozó eléri a maximális lehetséges értékét). Másik két rendszereljárás az Exit és a Halt. Az elsővel kiléphetünk az aktuális ciklusból, míg a második az egész program futását fejezi be.

A with utasítás

Az utolsó ciklusokban használható utasítás, amellyel itt bővebben foglalkozunk, sajátos a maga nemében. Jelenleg a Pascalon kívül csak a BASIC nyelvben található meg, és a Delphi programozók egyik nagyon hasznos utasítása.

Használatával rengeteg gépeléstől kímélhetjük meg magunkat. Amikor egy rekord típusú változót vagy objektumot használunk, akkor nevének többszöri ismételgetését feleslegessé lehet tenni a with utasítás használatával. Például, a rekord beállításához ezt az utasítássorozatot lehet használni:

```
type
  Date = record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;
var
  BirthDay: Date;

begin
  BirthDay.Year := 1997;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
end.
```

Ugyanezt a with használatával egyszerűbben is meg lehet fogalmazni:

```
begin
  with BirthDay do
  begin
    Year := 1995;
    Month := 2;
    Day := 14;
  end.
```

Ezzel a megközelítési módszerrel a Delphi programjaiban elemekre és egyéb osztálytípusokra lehet hivatkozni. Például a legutóbbi példánkat átírhatjuk a with utasítás használatával:

```
procedure TForm1.WhileButtonClick(Sender: TObject);
var
  I: Integer;
begin
  with ListBox1.Items do
  begin
    Clear; // shortcut
    Randomize;
    I := 0;
    while I < 1000 do
    begin
      I := I + Random (100);
      // shortcut:
      Add ('Random Number: ' + IntToStr (I));
    end;
  end;
B;
```

Általában amikor elemekkel és osztályokkal dolgozuk, a forráskód egy részét nyugodtan kihagyhatjuk, főleg az egymásba ágyazott mezőket. Például, ha meg kell változtatnunk egy form rajzolására használt toll vastagságát és színét, úgy a következő programrészletet alkalmazhatjuk:

```
Form1.Canvas.Pen.Width := 2;
Form1.Canvas.Pen.Color := clRed;
```

Vagy még egyszerűbben:

```
with Form1.Canvas.Pen do
begin
  Width := 2;
  Color := clRed;
end;
```

Bonyolultabb utasítássorozatok írásánál a with utasítás eredményesebben alkalmazható, rengeteg helyet megspórol, és feleslegessé teszi az ideiglenes változók létrehozását. Hátrányaként meg kell említeni, hogy a forráskódot

nehezebben értelmezhetőbbé teszi, főleg ha több, hasonló tulajdonsággal rendelkező objektummal dolgozunk.

További hátrány, hogy a with használatával nehezen felderíthető, a fordításnál nem jelentkező hibákat produkálhatunk. Például:

```
with Button1 do
begin
  Width := 200;
  Caption := 'New Caption';
  Color := clRed;
end;
```

Az utasítás végrehajtása után a gomb Caption és Width beállítása megváltozik, de a gomb színe helyett a formé változik meg! Ez azzal magyarázható, hogy a TButton elemnek nincsen Color nevű tulajdonsága, de a formnak van, így annak ilyen nevű tulajdonsága változik meg. Ha az előbbi helyett a következőt írnánk:

```
Button1.Width := 200;
Button1.Caption := 'New Caption';
Button1.Color := clRed; // error!
```

akkor a fordító hibaüzenettel leállna. Általánosságban mondhatjuk, hogy mivel a with utasítás új azonosítókat hoz létre az adott területen, így mi a már meglévőket biztonságosan elrejthetjük, vagy rossz értékadásokat hozhatunk létre ugyanabban az érvényességi körben (ahogyan ennek a programrészletnek az első verziójában láthattuk). Eme hátrányok ellenére is töltsünk el pár percet a with utasítás tanulmányozásával, mert egyes esetekben nagyon hasznos lehet az ismerete, és a forráskód olvashatóságát is nagy mértékben javíthatja. Ettől függetlenül, lehetőleg ne használjuk többszörös utasítások felépítésében, mint:

```
with ListBox1, Button1 do...
```

Az ezt követő programrészlet nagy valószínűség szerint eléggé bonyolult lehet, mert minden egyes elemnél el kellene gondolkodni azon, hogy melyikhez is tartozik, a megfelelő tulajdonságok és a with utasítás után álló elemek sorrendje alapján.

Tipp

Az olvashatóságnál maradva, a Pascalban nincs endif vagy endcase utasítás, ezért az if és case szerkezeteket egyaránt end zárja. Ha ezek az end utasítások egymás után találhatóak, akkor az könnyen ronthatja a forráskód olvashatóságát, helyzetükből azonban következtethetünk, hogy melyik részhez is tartoznak. Egy elfogadott megoldása ennek a problémának az, ha az összes end után a megjegyzésben kiírjuk, hogy mit is zár le.

Eljárások és függvények

Egy másik, a Pascal által bevezetett és kiemelten fontos fogalom a rutin. A Pascalban a rutin két formában fordulhat elő: eljárás (procedure) és függvény (function). Az egyetlen különbség az, hogy a függvénynek van, míg az eljárásnak nincsen visszatérési értéke.

Íme egy eljárás, és két egymástól alig különböző függvény meghatározása:

```
procedure Hello;
begin
  ShowMessage ('Hello world!');
end;

function Double (Value: Integer) : Integer;
begin
  Double := Value * 2;
end;

// or, as an alternative
function Double2 (Value: Integer) : Integer;
begin
  Result := Value * 2;
end;
```

A függvényben a visszatérési érték meghatározására szolgáló Result változó alkalmazása kezd egyre elterjedtebbé válni, és érthetőbbé is teszi a programot. Miután létrehoztuk a rutinokat, a következőképpen hivatkozhatunk rájuk:

```
Hello;
X := Double (Y);
Hello;
Z := Double2 (X);
```

Minden programozó ismeri az eljárások és függvények kezelésének alapjait, ezért most ezekkel kapcsolatos, de kicsit haladóbb dolgokkal foglalkozunk.

Új!

Amikor egy létező Delphi függvényt vagy VCL metódust hívunk, mindig emlékeznünk kell a paramétereinek számára és típusára. A Delphi megpróbál segíteni nekünk azzal, hogy az eljárás nevének begépelése és a zárójel nyitását követően néhány paramétert tanácsol. Ahogyan azt már a második fejezetben említettük, ez az újfajta szolgáltatás a Code Parameters névre hallgat, és az új Code Insight Technology (CIT) része.

Cím szerinti paraméterátadás

A Pascal rutinok két módon, érték szerint és cím szerint vehetnek át paramétereket. Ez utóbbi esetben nem a paraméter értéke, hanem csak egy hivatkozás másolódik be a rutin vermébe (a program ettől gyorsabban fut). Ez lehetővé teszi, hogy értékét az eljáráson vagy függvényen belül megváltoztassuk. A paraméter referenciaként való átadását a paraméter előtti var kulcsszóval jelölhetjük.

Megjegyzés

Ez a technika a legtöbb programozási nyelvben alkalmazható. Igaz, nincsen benne a C nyelvben, de a C++ már tartalmazza, ahol a & (referencia átadás) jelet alkalmazhatjuk. A Visual Basic programozási nyelvben az összes, nem ByVal-ként átadott paraméter hivatkozásként adódik át.

Egy példa:

```
procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
end;
```

Ebben az esetben a paraméter, mint kiindulóérték adódik át az eljárásnak, és tartalmazni fogja a visszatérési értéket is az eljárás lefutása után. Ha azt írjuk, hogy:

```
var
  X: Integer;
begin
  X := 10;
  DoubleTheValue (X);
```

akkor az X változó eredeti értéke 20 lesz, mivel a függvény az X változó kezdeti értékét veszi figyelembe, és azt veszi kiindulási értéknek.

Paraméterek hivatkozásként való átadása csak az előre meghatározott típusok, régi stílusú stringek és nagyobb rekordok esetén van értelme. A Delphi objektumok, törvényszerűen, mindig változatlanul adódnak át, mivel maguk is hivatkozások. Ebből kifolyólag egy objektum referenciaként való átadása nem túl értelmes cselekedet (néhány igen speciális és különleges esetet kivéve), ugyanis az egy „hivatkozás a hivatkozásra" átadásának felel meg. A Delphi hosszú stringjei ettől egy kissé eltérően működnek: ezek hivatkozásként

viselkednek, de ha több string is hivatkozik ugyanarra a helyre a memóriában, és az egyiket ezek közül megváltoztatjuk, akkor a frissítés előtt a teljes string másolásra kerül. A paraméterként átadott hosszú string csak a memóriafelhasználásban és a végrehajtás sebességében viselkedik úgy, mint egy hivatkozás.Ha módosítjuk az eredeti stringet, akkor annak tartalma változatlan marad. Ennek ellenére, ha a hosszú stringet, mint hivatkozást adjuk át az eljárásnak, akkor az eredeti tartalmát változtathatjuk meg.

Megjegyzés

A Delphi 3-ban egy újfajta paraméter is megjelenik, az out. Az out paraméternek nincsen kezdeti, csak visszatérési értéke. Ezt csak a COM eljárásokban és függvényekben szabad használni; általában jobb, ha a hagyományos var paraméterekre hagyatkozunk. Az out paraméter annak ellenére, hogy nincs kezdeti értéke, teljesen úgy viselkedik, mintha rendes var paraméter lenne.

Konstans paraméterek

A hivatkozási paraméterek mellett konstans paramétereket is használhatunk. Mivel a konstans paraméternek nem adhatunk meg új értéket a rutinon belül vagy kívül, a fordító optimalizálni tudja a paraméter átadását. A fordító választhatja a hivatkozásnak megfelelő módszert (a C++ nyelvben ezt konstans hivatkozásnak nevezik), de ettől függetlenül annak viselkedése nem változik meg, hiszen konstansról van szó.

Ha a következő (elég buta) utasítássorozatot próbáljuk lefordítani, akkor a Delphi hibaüzenetet jelenít meg:

```
function DoubleTheValue (const Value: Integer): Integer;
begin
  Value := Value * 2;        // compiler error
  Result := Value;
end;
```

Nyitott tömb paraméterek

A Pascal eljárásaiban és függvényeiben, a C-vel ellentétben, mindig előre meghatározott számú paramétert kell használni. Ennek ellenére, a nyitott tömbök használatával ezt a megszorítást meg lehet kerülni.

A nyitott tömb paraméter egy nyitott tömb típus. Ez azt jelenti, hogy megadjuk a paraméter típusát, de nem adjuk meg (mert nem tudjuk, vagy nem akarjuk), hogy az hány elemből áll. Ezt a következőképpen határozhatjuk meg:

```
function Sum (const A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(A) to High(A) do
    Result := Result + A[I];
end;
```

A High(A) használatával megtudhatjuk, hogy hány elemet tartalmaz a tömb. Vegyük észre azt is, hogy létezik visszatérési érték, és azt átmeneti változóként is használjuk. A fentebb említett eljárást Integerekből álló tömbparaméter átadásával hívhatjuk meg:

```
X := Sum ([10, Y, 27*I]);
```

Ha rendelkezünk már Integer típusú elemekből álló tömbbel, akkor azt közvetlenül átadhatjuk a nyitott tömböt váró eljárásunknak, vagy átadhatjuk annak egy részét a Slice függvény használatával (ahogyan azt a második paraméter mutatja). Íme egy példa, ahol egy teljes tömböt adunk át paraméterként:

```
var
  List: array [1..10] of Integer;
  X, I: Integer;
begin
  // initialize the array
  for I := Low (List) to High (List) do
    List [I] := I * 2;
  // call
  X := Sum (List);
```

Ha a tömbnek csak egy részét kívánjuk átadni paraméterként, akkor a Slice függvényt a következőképpen hívjuk meg:

```
X := Sum (Slice (List, 5));
```

Ezeket a kódrészleteket ebben a szakaszban, az OpenArr példában találhatjuk meg (lásd 4.19 ábra).

Változtatható típusú nyitott tömb paraméterek

A típussal rendelkező nyitott tömbök mellett, a Delphi lehetővé teszi a változtatható típusú, illetve típus nélküli nyitott tömbök létrehozását és használatát. Ezek a speciális fajtájú tömbök nem meghatározott számú értékkel rendelkeznek, amely őket különösen hasznossá teszi a paraméterek átadásában.

Gyakorlatilag a konstans tömbbel nem meghatározott számú, különböző típusú elemekből álló tömböt adhatunk át paraméterként. Például, itt van a Format függvény meghatározása (az előző fejezetekben már áttekintettük, hogy hogyan kell használni, így itt most a meghatározása nélkül tárgyaljuk):

```
function Format (const Format: string;
  const Args: array of const): string;
```

A második paraméter egy nyitott tömb, amely előre meg nem határozott számú értéket kap. Ezt a függvényt a következőképpen hívhatjuk meg:

```
N := 20;
S := 'Total:';
Label1.Caption := Format ('Total: %d', [N]);
Label2.Caption := Format ('Int: %d, Float: %f', [N,
    12.4]);
Label3.Caption := Format ('%s %d', [S, N * 2]);
```

Figyeljük meg, hogy paraméterként konstans értéket, változót és kifejezést is megadhatunk. Az ehhez hasonló függvény meghatározása elég könnyű, az egyetlen nehézséget annak „megkomponálása" jelentheti. A változó-típusú tömb értékei a TVarRec típus elemeivel kompatibilisek.

A Delphi 3-ban használható TVarRec rekord a következő szerkezettel rendelkezik:

```
type
  TVarRec = record
    case Byte of
      vtInteger:    (VInteger: Integer; VType: Byte);
      vtBoolean:    (VBoolean: Boolean);
      vtChar:       (VChar: Char);
      vtExtended:   (VExtended: PExtended);
      vtString:     (VString: PShortString);
      vtPointer:    (VPointer: Pointer);
      vtPChar:      (VPChar: PChar);
      vtObject:     (VObject: TObject);
      vtClass:      (VClass: TClass);
      vtWideChar:   (VWideChar: WideChar);
      vtPWideChar:  (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:   (VCurrency: PCurrency);
      vtVariant:    (VVariant: PVariant);
      vtInterface:  (VInterface: Pointer);
  end;
```

Minden egyes rekordnak van egy VType típusú mezője, annak ellenére, hogy ez nehezen látható, hiszen az csak egyszer lett létrehozva az aktuális integernagyságú adattal együtt.

Megjegyzés

Ne keverjük össze a TVarRec rekordot a Variant típus által közvetlenül alkalmazott TVarData rekorddal, ugyanis mindkettő különböző szerkezettel és felhasználási területtel rendelkezik, továbbá nem kompatibilisek egymással. Még a lehetséges típusok listája sem egyezik meg, mivel a TVarRec Delphi adattípusokat, míg a TVarData OLE adattípusokat tartalmazhat.

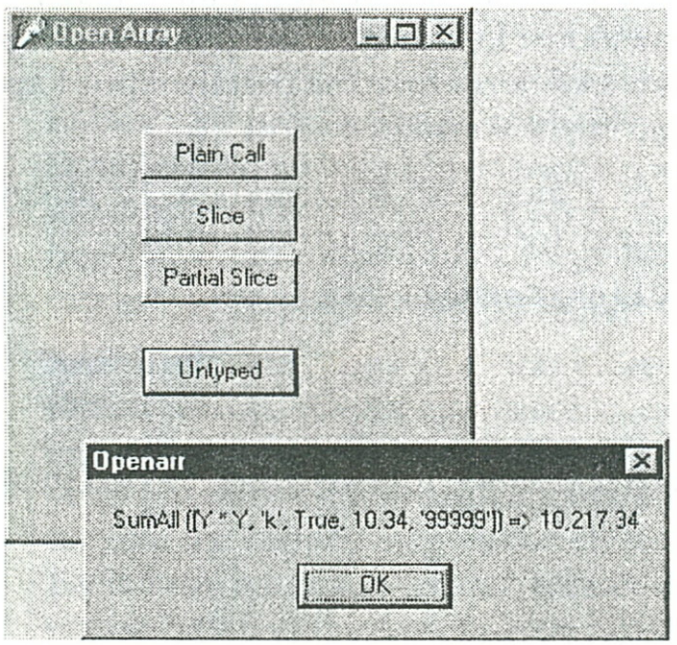
Ezt felhasználva olyan függvényeket írhatunk, amelyek képesek különböző típusú adatokon műveleteket végezni. A SumAll függvénnyel például különböző típusú elemek összegét számoltathatjuk ki, az igaz logikai értékekhez egyet adhatunk, stringeket számokká, a karaktereket pedig a nekik megfelelő értékké alakíthatjuk át. Az eljárás alapját a case utasítás képezi, elég egyszerűen, annak ellenére, hogy számos pointerre hivatkozunk menet közben:

```
function SumAll (const Args: array of const): Extended;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(Args) to High (Args) do
    case Args [I].VType of
      vtInteger: Result :=
        Result + Args [I].VInteger;
      vtBoolean:
        if Args [I].VBoolean then
          Result := Result + 1;
      vtChar:
        Result := Result + Ord (Args [I].VChar);
      vtExtended:
        Result := Result + Args [I].VExtended^;
      vtString, vtAnsiString:
        Result := Result + StrToIntDef ((Args
                    [I].VString^), 0);
      vtWideChar:
        Result := Result + Ord (Args [I].VWideChar);
      vtCurrency:
        Result := Result + Args [I].VCurrency^;
    end; // case
end;
```

Ezt tartalmazza az OpenArr példaprogram, amely a SumAll függvényt hívja meg, amikor egy megadott gombra kattintunk:

```
procedure TForm1.Button4Click(Sender: TObject);
var
  X: Extended;
  Y: Integer;
begin
  Y := 10;
  X := SumAll ([Y * Y, 'k', True, 10.34, '99999']);
  ShowMessage (Format (
    'SumAll ([Y*Y, ''k'', True, 10.34, ''99999'']) =>
                                          %n', [X]));
end;
```

A függvényhívás eredményét és az OpenArr példaprogram formját a 4.19. ábrán láthatjuk.



4.19. ábra *Az OpenArr példaprogram formja, az Untypesd gomb megnyomásakor jelentkező üzenetablakkal*

A Delphi alprogram-hívási lehetőségei

A Delphi 32 bites verziójában egy új paraméterátadási módszer jelent meg, amelyet fastcall-nak nevezünk. Ennek lényege az, hogy legfeljebb három paramétert a CPU regisztereiben helyezhetünk el, ami által a függvényhívás jelentősen felgyorsul. A fastcall metódus hívását a Delphi 3-ban a register kulcsszóval jelöljük.

Az egyetlen felmerülő probléma az, hogy az eredeti módszer és a fastcall metódus nem kompatíbilis a Windows operációs rendszerrel: a Win32 API függvények az stdcall módszerrel hívhatóak, amely az eredeti Win16 API Pascal hívási eljárás és a C nyelv cdecl hívási metódusának ötvözete.

Általában nincs olyan ok, amely a fastcall metódus használatát akadályozná, kivéve, ha külső Windows hívásokat használunk, vagy Windows callback függvényt hozunk létre. Még ennek a fejezetnek a vége előtt példát fogunk látni a stdcall hívás használatára. A különböző hívási módszerekről rengeteg információt találhatunk a Delphi helpjében, a „calling conventions" címszó alatt.

Mi is az a metódus?

Ha már dolgoztunk Delphi alatt, vagy már olvastunk Delphivel foglalkozó dokumentációt, bizonyára találkoztunk már a metódus (method) kifejezéssel. A metódus nem más, mint egy speciális függvény vagy eljárás, amely egy meghatározott adattípushoz vagy osztályhoz tartozik. A Delphin belül, valahányszor egy eseményt kezelünk le, egy metódust kell meghatároznunk, ami legtöbbször egy eljárás. Általában a metódus kifejezés mind az osztályokhoz kapcsolódó eljárásokra, mind pedig a függvényre vonatkozik.

Már láttunk számos metódust ebben és a megelőző fejezetekben. Ez itt egy üres metódus, amit a Delphi fűz hozzá egy frissen létrehozott formhoz:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  {here goes your code}
end;
```

A metódusokkal bővebben a következő fejezetben foglalkozunk, az osztályok szerepe kapcsán.

Elődeklarációk

Bármilyen azonosítót is használunk a programban, a fordítónak tudnia kell, hogy az mire hivatkozik, milyen paramétereket használ, és így tovább. Ebből kifolyólag kell meghatározni az azonosítónak a programban betöltött szerepét, még mielőtt egyáltalán elkezdenénk használni. Néhány esetben azonban ez nem alkalmazható. Például, ha az A eljárás meghívja a B eljárást és a B felhasználja az A eljárást, akkor mindenféleképpen meg fogunk hívni egy olyan rutint, amelyet még nem deklaráltunk.

Ennek elkerülése érdekében, a forward kulcsszó használatával a program elején megadhatjuk az eljárás vagy függvény minden tulajdonságát anélkül, hogy annak teljes kódrészletét mellékelni kellene:

```
procedure Hello; forward;
```

A későbbiekben természetesen meg kell adni az eljárás teljes forráskódját, de azt már a teljes meghatározás előtt meg lehet hívni. Íme egy buta példa, csak a forward használatának szemléltetésére:

```
procedure DoubleHello; forward;

procedure Hello;
begin
  if MessageDlg ('Do you want a double message?',
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    DoubleHello
  else
    ShowMessage ('Hello');
end;

procedure DoubleHello;
begin
  Hello;
  Hello;
end;
```

Ebben a programban többszörös egymásra hivatkozás található: a DoubleHello a Hello-t hívja meg, és a Hello meghívhatja a DoubleHello-t. Természetesen léteznie kell egy olyan utasításnak, amely ezt a láncot megszakítja, hogy a verem túlcsordulást megelőzzük. Ezt a példát találhatjuk meg kisebb módosításokkal a DoubleH példaprogramban is.

Annak ellenére, hogy az eljárás előre deklarálása nem általános a Delphi programok között, létezik egy hasonló eset, amely többször előfordulhat. Amikor egy eljárást vagy függvényt egy unit (a következő fejezetben foglalkozunk vele) interface-én keresztül határozunk meg, azt a program forwardolt meghatározásnak tekinti, még akkor is, ha a forward kulcsszót nem is használtuk. Ezért sem lehet egy eljárás törzsét az interface-ben leírni. Ugyanakkor egy uniton belül, az implementation részben deklarálni kell az összes interface-ben megadott eljárást és függvényt.

Ugyanez vonatkozik a Delphi által automatikusan létrehozott osztályokon belüli egyes metódusok meghatározására (azután, hogy a formhoz, vagy annak eleméhez egy eseményt rendeltünk). A TForm osztályban található eseménykezelők ugyancsak előre deklaráltak: az aktuális programrészletet a unit implementációs része adja. Az itt bemutatott programrészlet egy korábban tárgyalt programból származik, és tartalmazza a Button1Click metódus deklarását is:

```
type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  end;
```

Külső deklarációk

Egy másik különleges fajtája az eljárások meghatározásának a külső deklaráció. Eredeti szerepe a külső, assembly nyelven írt programok Pascal alá fűzésében volt, most a Windows külső, a DLL-ben (Dynamic Link Library) található függvényeinek hívására alkalmazható. A Delphiben számos, ehhez hasonló Windows meghatározás található:

```
// forward declaration
function LineTo (DC: HDC; X, Y: Integer): BOOL;
stdcall;

// external reference
function LineTo; external 'gdi32.dll' name 'LineTo';
```

A LineTo ilyen módon való meghatározása azt jelenti, hogy maga a függvény a GDI32.DLL dinamikus könyvtárban található (az egyik legfontosabb Windows rendszerkönyvtár) azzal a névvel, amit mi a programunkban használunk. A külső meghatározáson belül, ha akarjuk, azt is megadhatjuk, hogy az általunk használt eljárás vagy függvény egy teljesen más nevet viselő DLL függvényre hivatkozzon.

Általában véve ritkán fordul elő, hogy a fent vázolt módon hivatkozzunk egy eljárásra, hiszen ezek legtöbbje megtalálható a Windows unitban, és sok más Delphi rendszeregységben is. Az egyetlen eset, amikor ehhez kell folyamodnunk, ha nem dokumentált Windows rendszerfüggvényt hívunk meg, illetve ha saját DLL-t használunk.

Megjegyzés

A Delphi 16 bites verziójában a külső deklarációk a könyvtárak kiterjesztés nélküli nevét, majd a név (mint a fenti programrészletben) vagy egy másik direktívát, legvégül pedig a DLL-en belüli sorszámot használták. A változtatás a könyvtárak hozzáférési módszerén hagyott nyomot: bár a Win32 még mindig számokkal hivatkozik a DLL függvények használatakor, a Microsoft cég saját bevallása szerint ezt a jövőben már nem fogja támogatni. Azt is vegyük észre, hogy a Windows unit helyettesíti a Delphi 16 bites verziójának részét képező WinProcs és WinTypes unitokat.

Eljárás-típusok

Egy ugyancsak egyedi Object Pascal jellemző az eljárás típusok létezése. Ezek olyan fejlett nyelvi eszközök, amelyeket csak néhány Delphi programozó fog teljes mértékben kihasználni. Mivel ezekkel a későbbi fejezetekben még foglalkozunk (főleg a metódusmutatóknál, amely egy Delphiben sűrűn használt technika), nem árt, ha itt és most gyorsan áttekintjük őket. Ha nem érezzük magunkat elég tapasztaltnak a bonyolultabb programozási technikák elsajátításához, hagyjuk ki ezt a szakaszt, s csak később térjünk vissza ide.

A Pascalban, létezik egy C-beli függvénymutatóhoz hasonló eljárás-típus. Az eljárás-típusok létrehozásakor meg kell adni a paraméterek listáját és a függvény esetében a visszatérési érték típusát is. Például, így határozhatunk meg egy eljárás-típust, amely paraméterként integer értékű hivatkozást fogad el:

```
type
  IntProc = procedure (var Num: Integer);
```

Ez az eljárás-típus teljesen kompatíbilis minden olyan eljárással, amely pontosan ugyanezeket a paramétereket használja (vagyis a C zsargont használva, ugyanaz a függvény szignatúrája). Íme egy példa a kompatíbilis eljárásra:

```
procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
end;
```

Figyelem!

A Delphi 16 bites verziójában az eljárásokat a `far` direktívával kell lefordítani, ha azokat aktuális értékként vagy eljárás-típusként kívánjuk használni.

Az eljárás-típusok két cél érdekében használhatóak fel: létrehozhatunk eljárás-típusú változókat vagy átadhatunk eljárás-típusokat – mint függvény-mutatót – más rutinoknak. Az előbbi típussal és eljárással a következő programot írhatjuk meg:

```
var
  IP: IntProc;
  X: Integer;
begin
  IP := DoubleTheValue;
  X := 5;
  IP (X);
end;
```

Ennek ugyanaz a hatása, mint a rövidebb verziójának:

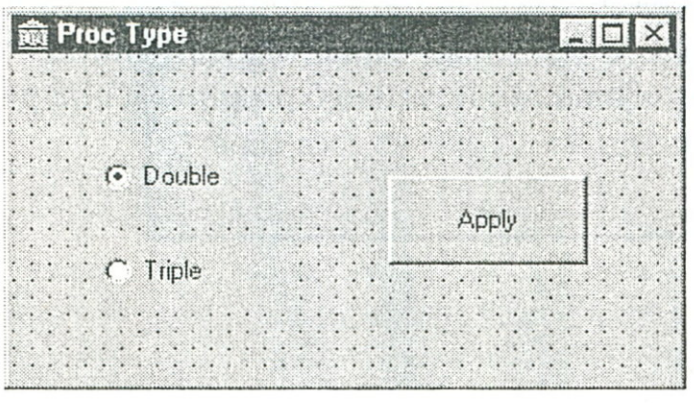
```
var
  X: Integer;
begin
  X := 5;
  DoubleTheValue (X);
end;
```

Az első változat ránézésre is sokkal bonyolultabb, ezért felmerülhet a kérdés, hogy miért kellene használni? Néhány esetben hasznos lehet, ha el tudjuk dönteni, hogy melyik függvényt kellene meghívni, és később meg is hívjuk azt. Lehetséges egy ennél jóval összetettebb példa megírása, amely ennek a módszernek a létjogosultságát támasztja alá. Ennek ellenére most érjük be egy egyszerűbb példával, amelyet ProcType-nak nevezünk. Ez a példa jóval bonyolultabb, mint az eddig látottak, hogy egy kicsit jobban hasonlítson a mindennapi életben előforduló programokra.

Egyszerűen hozzunk létre egy teljesen üres projektet, majd két rádiógombot, egy nyomógombot és két feliratot a formon, ahogy azt a 4.20. ábra mutatja. Egy eljárást használjunk a paraméterként megadott érték duplázására. Ez az eljárás nagymértékben hasonlítani fog ahhoz, amit korábban ebben a fejezetben láthattunk. A második eljárás, a TripleTheValue, a nevéhez híven a paraméterként megadott értéket háromszorozza meg:

```
procedure TripleTheValue (var Value: Integer);
begin
  Value := Value * 3;
  ShowMessage ('Value tripled: ' + IntToStr (Value));
end;
```

Mindkét eljárás kiírja, hogy mi történik bennük, és hogy honnan lettek meghívva. Ez az egyszerű nyomkövetési módszert máshol is eredményesen alkalmazhatjuk, ha ki akarjuk deríteni, hogy adott programrészek mikor és milyen változói környezetben hajtódnak végre.



4.20. ábra *A ProcType példa formja*

Minden esetben, ha megnyomjuk az Apply feliratú gombot, a két eljárás közül az egyik végrehajtódik, attól függően, hogy melyik rádiógomb van bekapcsolt állapotban. Természetesen, ha a formon csak két rádiógomb található, akkor egyszerre csak az egyik lehet aktív. Ennek ellenőrzését az Apply gomb OnClick eseményébe lehet beépíteni. Az eljárás-típusok használatához nézzünk inkább egy hosszabb, de érdekesebb megközelítést. Ebben a programban ahányszor egy rádiógombra kattintunk a megfelelő eljárást adjuk az IP változó értékéhez:

```
procedure TForm1.DoubleRadioButtonClick(Sender:
TObject);
begin
  IP := DoubleTheValue;
end;
```

Amikor a a nyomógombra kattintunk, akkor a változóban tárolt eljárás végrehajtódik:

```
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
  IP (X);
end;
```

Annak érdekében, hogy az IP és az X változót a három különböző függvény is elérje, az egész formon belül láthatóvá kell tennünk, azaz nem hozhatjuk őket létre egy metóduson belül, hanem a változót a form meghatározásában helyezzük el:

```
type
  TForm1 = class(TForm)
    ...
  private
    { Private declarations }
    IP: IntProc;
    X: Integer;
  end;
```

A következő fejezetben egészen pontosan meg fogjuk érteni, hogy ez mit is jelent, de egyelőre az osztály típushoz a Delphi által létrehozott kódrészletet kell módosítani a fenti módon, és hozzá kell adni az eljárás-típus meghatározását, úgy ahogy az előbb tettük. Ezeknek a változóknak a rendes értékekkel történő inicializálásához megváltoztathatjuk a form OnCreate eseménykezelőjét (az eseménykezelőt az Object Inspectorból választhatjuk ki a form aktivizálása után, vagy egyszerű duplakattintással). Segítség lehet még, ha tanulmányozzuk a CD példaprogramjait és leírásait a dolog megértése érdekében.

Windows callback függvények

Az eljárástípusok második fő felhasználási területe az API függvények számára callback (visszahívás) biztosítása. Először is, mi az a callback? A lényeg az, hogy néhány API függvény ugyanazt a műveletet sok rendszerelemen, például bizonyos típusú ablakokokon végzi el. Ezek a függvények, más néven sorszámozott függvények (enumerated functions), paraméterként egy eljárás-típusban megadott műveletet kérnek, amit aztán a kijelölt elemeken elvégeznek. A Windows a callback függvényt más esetekben is alkalmazza, de számunkra ez most nem érdekes, mivel csak az egyszerű esetekkel foglalkozunk:

Vegyük például az EnumWindows API függvényt, amelynek a következő szerkezete van (a Win32 Helpjéből másoltuk ki):

```
BOOL EnumWindows(
  WNDENUMPROC lpEnumFunc,  // address of callback
                           // function
  LPARAM lParam // application-defined value
  );
```

Természetesen, ez egy C nyelven írt meghatározás. A WINDOWS.PAS fájlba nézve megtalálhatjuk a megfelelő Pascal nyelven írt meghatározását:

```
function EnumWindows (
  lpEnumFunc: TFNWndEnumProc;
  lParam: LPARAM): BOOL; stdcall;
```

A Súgót átböngészve azt kell találnunk, hogy a paraméterként átadott függvénynek a következő típusúnak kell lennie (megint C-ben):

```
BOOL CALLBACK EnumWindowsProc (
  HWND  hwnd, // handle of parent window
  LPARAM  lParam // application-defined value
  );
```

Ez a következő Delphi eljárástípusnak felel meg:

```
type
  EnumWindowsProc = function (Hwnd: THandle;
    Param: Pointer): Boolean; stdcall;
```

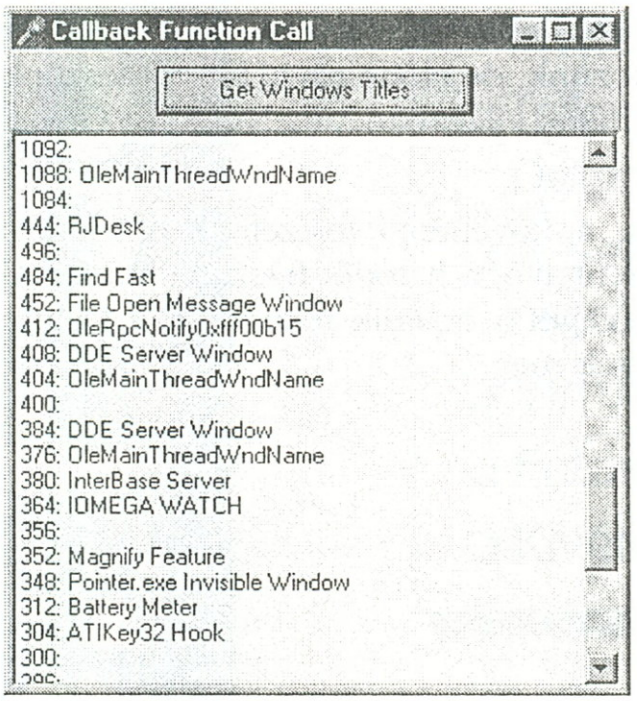
Az első paraméter tartalmazza az ablakok handle-jeit, míg a második az, amit mi adtunk át az EnumWindows függvény meghívásakor. A Pascalban a TFNWndEnumProc típus nincsen megfelelően meghatározva, sőt, mutatóként viselkedik. Ez pedig azt jelenti, hogy egy megfelelő paraméterekkel rendelkező függvényt kell biztosítanunk, amit mutatóként kell használni, azaz a címét kell átadni ahelyett, hogy végrehajtanánk. Sajnos eme csűrt-csavart módszer miatt a fordító nem nyújt semmiféle segítséget paraméterek ellenőrzéséhez.

Íme a megfelelő meghatározása egy olyan kompatíbilis függvénynek, amely egy ablak fejlécét egy stringbe olvassa be, majd azt egy megadott form listájához adja:

```
function GetTitle (Hwnd: THandle; Param: Pointer):
Boolean; stdcall;
var
  Text: string;
begin
  SetLength (Text, 100);
  GetWindowText (Hwnd, PChar (Text), 100);
  FormCallBack.ListBox1.Items.Add (
    IntToStr (Hwnd) + ': ' + Text);
  Result := True;
end;
```

A formot szinte teljesen eltakarja a hozzátartozó list box, csak egy kis rész marad üresen fenn, ahol egy nyomógomb található. A nyomógombon való kattintással az EnumWindows API függvényt hívjuk meg a GetTitle paraméterrel:

```
procedure TFormCallback.BtnTitlesClick(Sender: TObject);
var
  EWProc: EnumWindowsProc;
begin
  ListBox1.Items.Clear;
  EWProc := GetTitle;
  EnumWindows (@EWProc, 0);
end;
```



4.21. ábra *A Callback példa kimenete, mely felsorolja az aktuális (látható és rejtett) ablakokat*

A függvényt úgy is meghívhattuk volna, hogy nem tároljuk el az értékét egy ideiglenes változóban, de a program működésének megértéséhez és áttekinthetőségéhez ez elengedhetetlen volt. A program hatása meglehetősen érdekes, nézzük csak meg a 4.21. ábrát. A Callback példaprogram az összes létező ablak listáját adja vissza. A legtöbbet nem is láthatjuk, hiszen más ablakok rejtik őket, illetve nincsen fejlécük.

Hogyan tovább?

Ebben a fejezetben az Object Pascal alapvető elemeivel ismerkedhettünk meg, és betekintést kaphattunk a nyelv és a Windows néhány olyan eszközének használatába, mint a handle-k, API függvényhívások, és a callback függvények.

Még nem foglalkoztunk a Delphi nyelvbe épített objektumkezeléshez kapcsolódó lehetőségekkel, hiszen azokról a következő fejezetben olvashatunk. Ugyancsak nem beszéltünk a program unitokra való szétbontásáról és más programozási technikákról.

Ahogyan azt már korábban már említettük, ez a fejezet főleg olyan programozók érdeklődésére tarthat számot, akik már rendelkeznek némi tapasztalattal a Pascal nyelvvel kapcsolatban. Nem tértünk ki részletesebben a for ciklusutasítás és az if utasítás használatára. Ha ezekkel nem vagyunk teljesen tisztában, akkor töltsünk el egy kis időt a Delphi Súgó fájlaival és egyéb, Pascal nyelvű szövegekkel.

Ez nem jelenti azt, hogy az itt használt kifejezéseket a lehető legpontosabban meg kell értenünk. A könyv még nagyon sok olyan példát tartalmaz, amely a Pascal nyelv ínyencségeit emeli ki. A további részekben az ebben a fejezetben érintett témákkal részletesebben foglalkozunk, de előtte rövid kitérőt teszünk az objektumorientált programozási technikák világában, amelyeket az Object Pascal szemszögéből vizsgáljuk meg.

Ez három fejezeten keresztül húzódik: a következő fejezet az objektumorientált programozás alapjainak van szentelve, a hatodik fejezetben a fejlettebb nyelvi eszközökkel foglalkozunk, végül a hetedik fejezetben a Delphi komponenskönyvtárát, a VCL-t ismerhetjük meg.

5. FEJEZET

Az Object Pascal mint Objektum-Orientált nyelv

E fejezet tartalma:

- Az Object Pascal Osztályok és objektumok
- A Delphi objektum hivatkozási modellje
- Információ elrejtése és a Delphi egységek
- A Self kulcsszó és az osztálymetódusok
- Virtuális metódusok
- Futásidejű típusinformációk
- Kivételkezelés

Az Object Pascal, mint objektumorientált nyelv

A létező és felhasználó által definiált adattípusok használata, és a programkód kis blokkokba – eljárásokba – szervezése: ezek a tradicionális Pascal programozás alapelemei. De a Pascal nyelv új változatai a programírás egy teljesen új megközelítését is lehetővé teszik, az úgynevezett objektumorientált programozást (OOP). Az OOP nyelvek általában három alapelvre épülnek: osztályokra, öröklődésre, és polimorfizmusra (vagy késői kötésre). A fejezet első részében azt nézzük meg, hogy ez a három alapelv hogyan jelenik meg az Object Pascalban.

Megjegyzés

Az OOP-vel kapcsolatos elméletről rövid összefoglalást ad az A függelék. Ha most találkozunk először OOP nyelvvel, akkor javasolt, hogy a fejezet elolvasása előtt nézzük át az A függeléket.

Delphi alkalmazásokat írhatunk anélkül is, hogy ismernénk az Object Pascal részleteit. Amikor új formot készítünk, új komponenst teszünk rá, vagy eseményeket kezelünk, a kapcsolódó kód legnagyobb részét a Delphi elkészíti nekünk. De ha ismerjük a nyelv részleteit és felépítését, az hozzásegíthet, hogy megértsük, mit csinál a Delphi pontosan, és ezen keresztül jobban elsajátíthatjuk a nyelvet. Más szempontból nem kell értenünk az OOP-hez, nem kell tudnunk, hogy mi az a virtuális metódus, vagy a polimorfizmus ahhoz, hogy Delphi alkalmazást készítsünk, de új komponens tervezésénél ez a tudás elengedhetetlen.

Osztályok és objektumok

Az osztály és az objektum két gyakran használt szó, de mivel sokszor hibásan használják, definiáljuk mindkettőt a félreértések elkerülése végett! Az osztály egy felhasználó által definiált adattípus, aminek van állapota (a megjelenése), és vannak műveletei (a viselkedése). Az osztálynak vannak belső adatai és metódusai, eljárások és függvények képében, és néhány, nagyon hasonló dolog általános jellemzőit és viselkedését írja le. Az osztályokat a programozó a forráskód elrendezésére használja, a fordító pedig az alkalmazás felépítésére.

Az objektum az osztály egy példánya, vagy más szóval egy, az adott osztály által definiált típusú változó. Az objektumok valójában egyedek, és amikor a program fut, akkor az objektumok memóriát foglalnak a belső értékeik számára. A kapcsolat az objektum és az osztály között ugyanolyan, mint a változó és a típus esetében.

Figyelem!

Sajnos néhány nyelvben és fejlesztőeszközben ez a különbség nem tiszta. A zavar fokozására a Borland Pascal fordító korábbi változatai az objektum kulcsszóval definiálták az osztályokat. Emiatt a régebb óta Pascalban programozók gyakran az objektum kifejezést használják az osztály helyett a típusokra, és az objektum példány kifejezéssel jelölik a tényleges objektumot.

Új osztály létrehozásának az Object Pascalban a következő a szintaxisa:

```
type
  MyNewClass = class
  end;
```

Természetesen, ez a kód nem túl hasznos, hiszen az osztálynak nincs egyetlen adata vagy művelete sem, úgyhogy adjunk hozzá néhány adatot:

```
type
  TDate = class
    Month, Day, Year: Integer;
  end;
```

Tipp

Delphiben az a konvenció él, hogy minden osztálynevet T betűvel kezdünk. Ez csak egy konvenció, hiszen a fordító számára a T ugyanolyan betű, mint bármi más, de annyira elterjedt, hogy ha ezt betartjuk, akkor a programunk sokkal érthetőbbé válik. A könyvben megpróbálunk ragaszkodni ehhez a szokáshoz. Másik konvenció, hogy az osztály szerzőjét – személyt vagy céget – egy két- vagy hárombetűs kóddal jelöljük. Ez általában a komponensek esetében él, és ezt a könyvben is így fogjuk használni, mert a Delphi nem tud két azonos nevű komponenst installálni.

A TDate osztályunk deklarációja nagyon hasonlít egy rekordéra. Valójában a szokásos módon is deklarálhatunk egy objektumot, és ugyanazzal a módszerrel hivatkozhatunk is a mezőire:

```
var
  ADay: TDate;
begin
```

```
  ... // object allocation is missing here
  ADay.Month := 7;
  ADay.Day := 12;
  ADay.Year := 1984;
end;
```

A dolgok akkor válnak érdekesebbé, ha metódusokat – függvényeket vagy eljárásokat – is készítünk, hogy az osztályhoz műveleteket rendeljünk:

```
type
  TDate = class
    Month, Day, Year: Integer;
    procedure SetValue(m, d, y: Integer);
    function LeapYear: Boolean;
  end;
```

Ezt a függvényt, és az eljárást a kódban ki kell fejtenünk, mert itt még csak azt jelöltük, hogy a TDate osztály részei. Erre az Object Pascal szintén a pontot használja, de most egy kicsit másként, itt a szintaxis Osztálynév.Metódus:

```
procedure TDate.SetValue(m, d, y: Integer);
begin
  Month := m;
  Day := d;
  Year := y;
end;

function TDate.LeapYear: Boolean;
begin
  if (Year mod 4 <> 0) then
    LeapYear := False
  else if (Year mod 100 <> 0) then
    LeapYear := True
  else if (Year mod 400 <> 0) then
    LeapYear := False
  else
    LeapYear := True;
end;
```

Miután megírtuk ezeket a metódusokat, a következő módon meg is hívhatjuk őket:

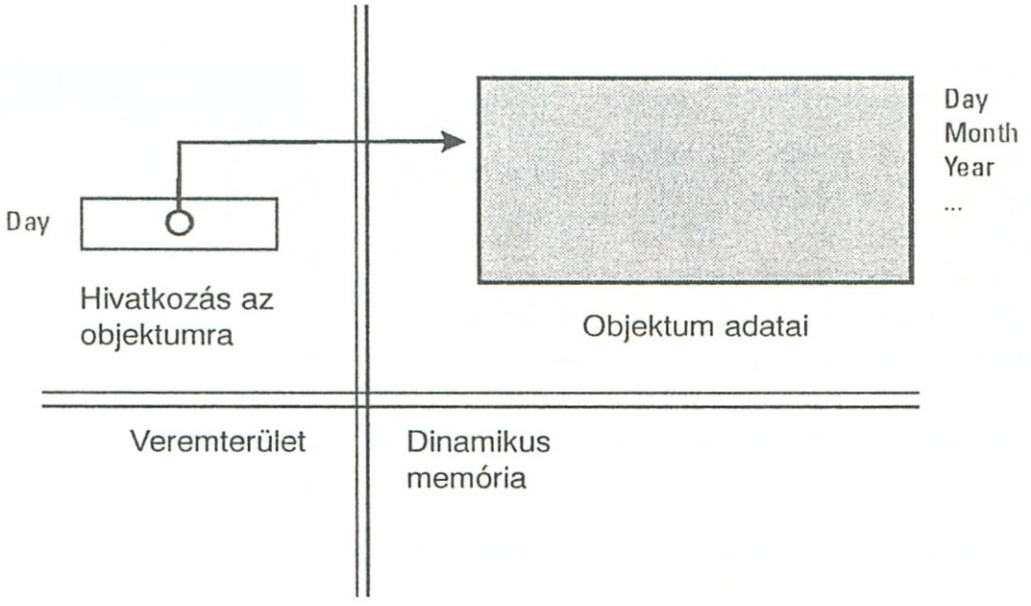
```
var
  ADay: TDate;
  Leap: Boolean;
begin
  ... // object allocation is still missing here
  ADay.SetValue (10, 10, 1997);
  Leap := ADay.LeapYear;
end;
```

A használt jelölés furcsa, de nagyon hasznos. Összetett eljárásokat írhatunk, mint amilyen a LeapYear (szökőév), és azután minden TDate objektumban hozzáférhetünk az értékéhez, mintha egyszerű adattípus lenne. Figyeljük meg, hogy az ADay.LeapYear kifejezés ugyanolyan, mint az ADay.Year, pedig a jelentésük különböző. Az első kifejezés egy függvényhívás, míg a második egy közvetlen adatelérés. Amint azt a következő fejezetben látni fogjuk, az Object Pascalban a tulajdonságok elérése is ugyanezzel a jelöléssel történik.

Bár a fenti kód tökéletes, mégsem működik, mert az Object Pascal egy objektum-hivatkozási modellt használ, ezért először létre kell hoznunk a TDate objektum egy példányát, ami a következő rész témája lesz.

A Delphi objektum-hivatkozási modellje

A legtöbb OOP nyelvhez hasonlóan a Class (osztály) kulcsszóval definiálhatunk új adattípust, új osztályt. Sok más nyelvben egy osztály típusú változó létrehozásakor keletkezik egy példány, egy objektum is. Az Object Pascal azonban egy objektum-hivatkozási modellen alapul, aminek az alapgondolata az, hogy az osztály minden példánya, mint például a Day a fenti kódrészletben, nem tárolja el az objektum értékét, csak egy hivatkozást, vagy mutatót, ami egy memóriaterületet jelöl, ahol az objektum tárolódik. Ennek a sémának az ábrázolása az 5.1. ábrán látható.



5.1. ábra *Az Object Pascal hivatkozási modelljének grafikus ábrázolása*

A hivatkozási modell nagyon sokoldalú, és könnyebben használható, mint sok más modell. Hasonló szemléletű OOP nyelv az Eiffel vagy a Java. Véleményünk szerint a hivatkozási modell átvétele volt a Delphi fejlesztő csapat egyik legjobb döntése.

Az egyetlen probléma ezzel a módszerrel, hogy amikor változót deklarálunk a következő módon:

```
var
  ADay: TDate;
```

nem készül el az objektum a memóriában, csak egy hely, ami az objektum-hivatkozást tárolni fogja – egyfajta mutató. Az objektum példányokat kézzel kell elkészítenünk, azoknál az osztályoknál mindenképpen, amiket mi definiáltunk. A formra rakott komponensek példányait a Delphi automatikusan elkészíti.

Hogy létrehozzuk az objektum egy példányát, meg kell hívnunk a Create metódust, ami egy konstruktor (különleges eljárás, ami az új objektumokat inicializálja, és memóriát foglal nekik, ahogy azt később láthatjuk). Vegyük észre, hogy a konstruktor az osztályra vonatkozik, nem az objektumra:

```
var
  ADay: TDate;
begin
  ADay := TDate.Create;
  ADay.Month := 7;
  ADay.Day := 12;
  ADay.Year := 1984;
end;
```

Honnan van a Create metódus? Ez a TObject osztály konstruktora, azé az osztályé, amiből az összes többi osztály öröklődik (az öröklésről ebben a fejezetben lesz szó). Másszóval ha azt írjuk, hogy:

```
type
  TDate = class
```

ugyanaz, mintha azt írnánk, hogy:

```
type
  TDate = class (TObject)
```

Természetesen, ha egyszer létrehoztunk egy objektumot, akkor azt meg is kell semmisíteni. Ezt a Free metódus meghívásával tehetjük meg. Ez is a TObject osztály metódusa. A következő példa egy objektum helyes létrehozását, felhasználását és megsemmisítését mutatja be:

```
begin
  // create
  ADay := TDate.Create;
  // use
  ADay.Month := 7;
  ADay.Day := 12;
  ADay.Year := 1999;
```

```
    if ADay.LeapYear then
      ShowMessage ('A leap year');
    // destroy
    ADay.Free;
end;
```

Ameddig az objektumokat akkor hozzuk létre, amikor kellenek, és megsemmisítjük őket, mikor már nem szükségesek, a hivatkozási modell probléma nélkül működik. Objektumot értékül adhatunk másik objektumnak, függvény paraméterként használhatunk objektumokat, és bármilyen más műveletet is elvégezhetünk velük.

Új!

Az objektum-allokálások megszüntetésének nyomon követése nem mindig egyszerű, de a Delphi 3 nyújt némi támogatást, ha a COM objektumokkal dolgozunk, ahogy az a következő, és a 21. fejezetben láthatjuk.

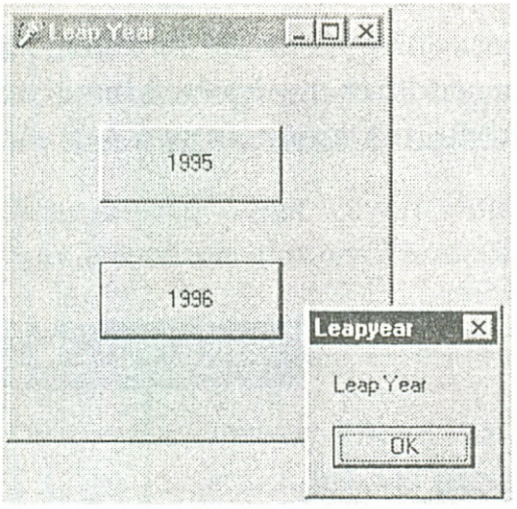
A TDate osztály felhasználása Delphiben

Már elég sok dolgot megnéztünk anélkül, hogy bármit ténylegesen kipróbáltunk volna, úgyhogy térjünk vissza a Delphihez. Gyűjtsük össze az eddig megírt TDate osztály kódrészleteket, és írjunk egy egyszerű programot, ami használja. Ezzel meggyőződhetünk arról, hogy amit írtunk működik, és különösen érdekes, hogy megbizonyosodjunk róla, hogy az objektum helyesen inicializálódik-e, vagy sem.

A LeapYear példa egy egyszerű formból áll, két gombbal, amiken két évszám szerepel címkeként. Amikor megnyomjuk valamelyik gombot, beállítjuk a TDate objektum értékét, és megnézzük, hogy 1997 (vagy 1996) szökőév volte. Itt következik az egyik gomb OnClick eseménykezelő kódja:

```
procedure TForm1.Btn1997Click(Sender: TObject);
begin
  ADay.SetValue (1, 1, 1997);
  if ADay.LeapYear then
    ShowMessage ('Leap Year')
  else
    ShowMessage ('Non Leap Year');
end;
```

A másik gomb kódja ugyanez, és az 5.2. ábrán látható eredményt idézi elő.



5.2. ábra *A LeapYear példa 1996 gombjának hatása*

Az ADay objektum egy új globális változó, amit így definiálhatunk:

```
var
  Form1: TForm1;
  ADay: TDate;
```

Ezt az objektumot még a gombok megnyomása előtt létre kell hozni. Erre általános megoldás a form OnCreate (készítéskor) eseményének használata:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ADay := TDate.Create;
end;
```

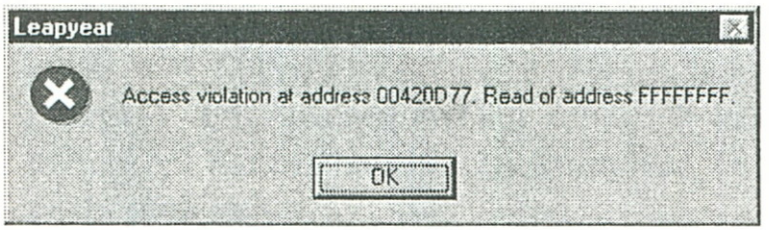
Ez a metódus lefut, mielőtt a felhasználó megnyomhatná a gombokat, még azelőtt, hogy a form vagy a többi komponens megjelenne a képernyőn. Ennek ellentéte az OnDestroy (megsemmisítéskor) esemény, ami azután fut le, miután a form eltűnt a képernyőről. Ezt a metódust felhasználhatjuk az objektumunk által használt memória felszabadítására:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  ADay.Free;
end;
```

Ameddig ismerkedünk a kóddal, próbáljuk ki a következőt, tegyük megjegyzésbe a példány előállítására használt kódot:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // ADay := TDate.Create;
end;
```

Ha most lefordítjuk a programot, nem kapunk fordítási hibát, de ha futtatjuk, akkor amint megnyomjuk akármelyik gombot, a rendszer egy kivételt hoz létre (ezzel jelezve, hogy nem létező mutatóra hivatkoztunk), amint az az 5.3. ábrán látható.



5.3. ábra *A kivétel, ami akkor keletkezik, ha nem hozzuk létre az objektum példányát*

Konstruktor készítése

Ahhoz, hogy memóriát foglaljunk az objektumnak, a Create metódust használtuk, de mielőtt ténylegesen használni tudnánk az objektumot, általában további inicializálás szükséges. Például a TDate osztály SetValue eljárását meg kell hívnunk, hogy működőképes legyen.

Erre nyújthat megoldást egy saját konstruktor, ami elkészíti az objektumot, és kezdőértéket is ad neki. Készíthetünk új verziót a Create metódusból, akár paraméterekkel is, vagy írhatunk teljesen új konstruktort, másik névvel. Ehhez írjunk egy új eljárást, akármilyen néven, akármennyi paraméterrel, de a `procedure` kulcsszó helyett a `constructor` szót használjuk.

A konstruktor különleges eljárás, mert amikor ezt egy osztályon használjuk, a Delphi automatikusan memóriát foglal le egy, abba az osztályba tartozó objektum számára. Ha viszont ezt a konstruktort egy objektumon használjuk, akkor csak a kód fut le memóriafoglalás nélkül, ezért ne futtassuk le egy inicializálatlan változó konstruktorát, hiszen memóriafoglalás csak akkor van, ha a konstruktort egy osztályon alkalmazzuk.

Saját konstruktor használatának legfontosabb célja az adatok inicializálása, mert ha egy inicializálatlan objektum metódusait meghívjuk, annak kiszámíthatatlan következményei lehetnek, akár futási hibák is. Ahelyett, hogy ezeket a hibákat várjuk, inkább alakítsunk ki megelőző technikát az elkerülésükre, egy ilyen technika lehet az adatok inicializálására a konstruktorok következetes használata.

Ugyanúgy, ahogy egy osztálynak lehet saját konstruktora, lehet saját destruktora (megsemmisítője) is. Ez egy olyan eljárás, amit a `destructor`

kulcsszóval deklarálunk, és ami felszabadíthat néhány erőforrást mielőtt az objektumunk megsemmisül. Ugyanúgy, ahogy a konstruktor meghívása lefoglalja a memóriát, a destruktor felszabadítja azt.

Destruktorok csak olyan objektumokhoz kellenek, amik keletkezésükkor, vagy létezésük alatt erőforrásokat foglaltak le.

Megjegyzés

A Destroy közvetlen hívása helyett célszerű a Free-t használni, ami csak akkor hívja meg a Destroyt, ha az objektum létezik, vagyis ha a rá mutató pointer nem nil. Jegyezzük meg azonban, hogy a Free nem állítja az objektum-mutatót nil-re, ezt nekünk kell megtennünk!

A TDate osztály konstruktorának elkészítéséhez a következő kódot használhatjuk:

```
type
  TDate = class
    Month, Day, Year: Integer;
    constructor Init (m, d, y: Integer);
    procedure SetValue(m, d, y: Integer);
    function LeapYear: Boolean;
  end;
```

Bár a példában a konstruktor az Init nevet viseli, bármilyen név megteszi. Érdemes általában a Create nevet használni, de az első példában szándékosan nem így tettünk, hogy megmutassuk, hogy a Create név semmiben nem különbözik az összes többitől. Az egyetlen lényeges elem a constructor kulcsszó. Ezután megírhatjuk a konstruktor kódját, ami a következő lehet:

```
constructor TDate.Init (m, d, y: Integer);
begin
  Month := m;
  Day := d;
  Year := y;
end;
```

Amikor ezzel készen vagyunk, módosíthatjuk a LeapYear példa inicializáló kódját, hogy az ADay memóriafoglalásával egy időben az objektum kezdőértékeit is beállítsa:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ADay := TDate.Init (1, 1, 1900);
end;
```

Objektumok a memóriában

Az előbb ismertetett módszert a Leap2 példában használjuk, ami a LeapYear példa új változata. Az új példa konstruktorokat definiál és használ, és hogy bemutassa, hogy a Delphi hogyan foglalja le a memóriát a konstruktorokkal, még az objektumok memóriabeli helyét is megmutatja. Ez nem olyan bonyolult: egyszerűen csak átdefiniáljuk az objektum értékét, a referenciát egész számmá, és kiírjuk. Íme a példakód:

```
ShowMessage (
  'Memory location for ADay: ' + IntToStr (Integer
      (ADay)));
```

Ez a kódrészlet akkor fut le, amikor az objektum először létrejön – vagyis a form OnCreate eseménykezelőjében, ami azt jelenti, hogy az üzenet ablak még a form előtt megjelenik –, valamint amikor az objektum értéke megváltozik. Valójában, amikor az egyik gombot megnyomjuk, akkor a program a SetValue metódust hívja meg, mint a LeapYear példában, de amikor a másik gombot nyomjuk meg, a program az Init konstruktort hívja meg az objektumon:

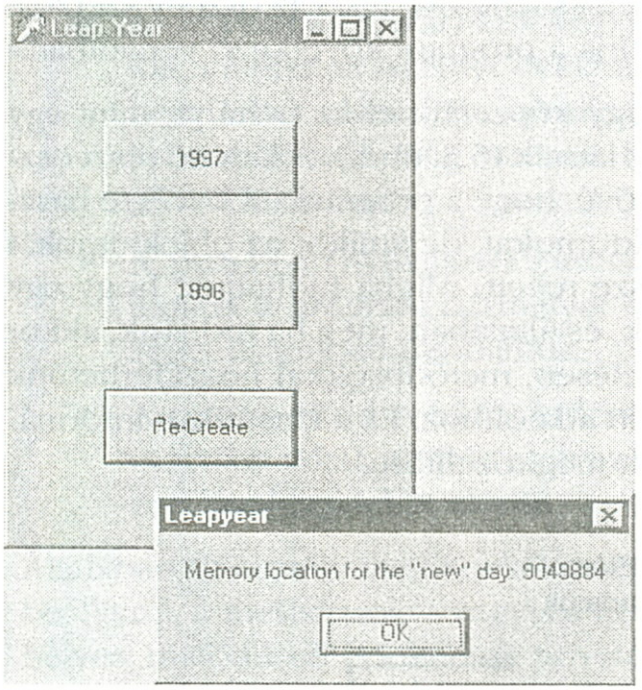
```
procedure TForm1.Btn1997Click(Sender: TObject);
begin
  ADay.Init (1, 1,1997);
  if ADay.LeapYear then
    ShowMessage ('Leap Year')
  else
    ShowMessage ('Non Leap Year');
  ShowMessage (
    'Memory location for ADay (after calling Init): ' +
    IntToStr (Integer (ADay)));
end;
```

Amint azt már említettük, ha a konstruktort az objektumon használjuk, akkor az csak a kódot futtatja le, minden különleges hatás nélkül. Az utolsó ShowMessage hívás megmutatja, hogy az objektum memóriabeli helye nem változik.

A form harmadik gombja, a Re-Create (újra létrehoz), új objektumot hoz létre a memóriában, és értékül adja azt az ADay változónak:

```
procedure TForm1.BtnCreateClick(Sender: TObject);
var
  NewDay: TDate;
begin
  NewDay := TDate.Init (1, 1, 1900);
  ShowMessage (
    'Memory location for the "new" day: ' +
    IntToStr (Integer (NewDay)));
  ADay.Free;
  ADay := NewDay;
end;
```

Ebben az esetben az objektum helye a memóriában megváltozik. Ha egymás után többször megnyomjuk ezt a gombot, láthatjuk, hogy csak néhány memória hely ismétlődik folytonosan, ugyanis a Delphi az első szabad memória területet adja az objektumnak, és ez lehet az a hely, amit éppen az előbb szabadítottunk fel az előző objektum megsemmisítésekor. Az 5.4. ábra mutatja a Re-Create gomb hatását.



5.4. ábra *A TDate objektum új helye a memóriában, a Leap2 példa utolsó gombjának megnyomása után*

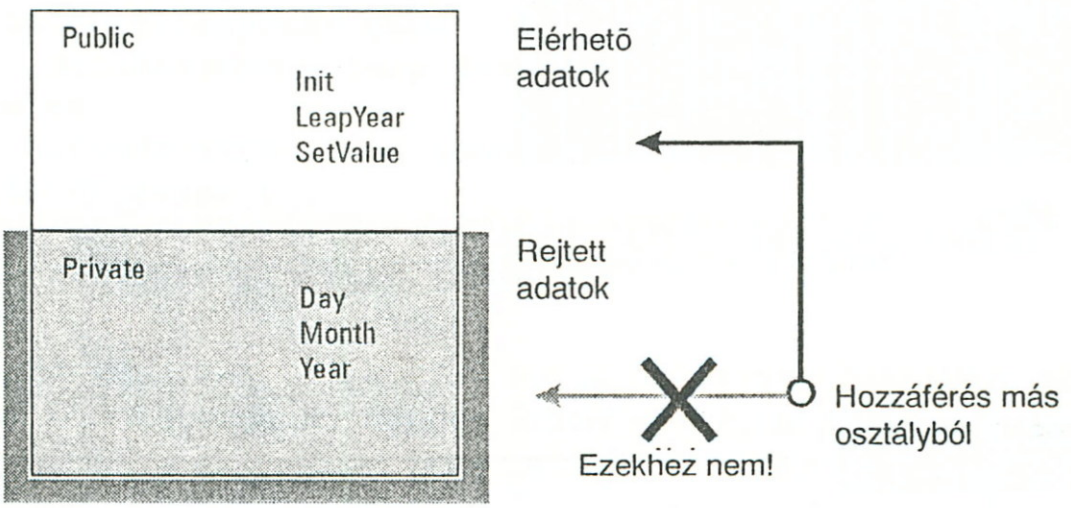
Megjegyzés

Amikor egy objektumnak értékül adunk egy másikat, akkor a Delphi csak memóriában lévő objektumra utaló hivatkozásokat másolja át az új objektumba. Ahhoz, hogy az objektum adatait is átmásoljuk egy másik objektum memóriaterületére, az összes mezőjét át kell másolnunk, vagy biztosítanunk kell egy eljárást, ami átmásolja a belső adatokat. Néhány VCL osztálynak van Assign metódusa, ami ezt a fajta másolást végzi.

Az osztályok és az információ-rejtés

Egy osztálynak akármennyi adata és akárhány metódusa lehet. Jó objektumorientált szokás szerint az adatokat el kell rejteni, vagyis be kell zárni az osztály belsejébe. Például amikor a dátumot használjuk, nincs értelme csak a napot állítani, mert az olyan furcsa értékekhez vezethet, mint például február harmincadika. Ha viszont az osztály belső részeit csak metódusok segítségével érjük el, az lecsökkenti a hibázási lehetőségeket, és lehetővé teszi az osztály szerzője számára, hogy az osztály belső részeit átírja a program következő verzióiban.

Az elrejtés elmélete nagyon egyszerű: úgy képzeljük el az osztályt, mint egy „fekete dobozt", amiből csak egy kis rész látszik (5.5. ábra). A látható részt osztály interfésznek hívjuk, és ez teszi lehetővé, hogy a program többi része használhassa az adott osztályba tartozó objektumokat, de amikor az objektumokat használjuk, a belső kód legnagyobb része rejtett. Aligha tudhatjuk, hogy egy objektumnak milyen belső adatai vannak, és általában, még ha tudnánk, akkor sem érhetnénk el közvetlenül. Természetesen, metódusokkal hozzáférhetünk az amúgy illetéktelen hozzáféréstől védett adatokhoz. Ez a klasszikus információ-rejtés elméletének objektum orientált megközelítése.



5.5. ábra

Az információ-rejtés grafikus ábrázolása

Elméletileg egy Pascal programozó számára ebben semmi új nincs, hiszen az OOP ugyanazokat az ötleteket követi a rendszerről, a típusdefiníciókról, és a biztonságról, amit a Pascal nyelv szorgalmazott. Az Object Pascalban két különböző konstrukció van a beágyazásra és a védelemre: az osztályok és az egységek. Az osztályokkal kapcsolatban van néhány hozzáférés specifikus kulcsszó, amiről a következő részben lesz szó.

Private, protected, public

Az Object Pascal átvette a C++-ból a három hozzáférést jelölő szót: ezek a private, a protected és a public. A negyedikről, a publishedről röviden a következő részben, részletesebben a következő fejezetben lesz szó. Még egy másik hozzáférés jelölőről, az OLE Automation által használt automatedről a 22. fejezetben olvashatunk.

- A `private` (privát) kulcsszó az osztály olyan mezőit és metódusait jelöli, amiket az osztályt deklaráló egységen (forrás állományon) kívülről nem lehet elérni. Ez némileg különbözik a C++-tól, ahol az osztály private részeit még azonos forrásban definiált másik osztályok sem érhették el.
- A `public` (nyilvános) kulcsszó olyan mezőket és metódusokat jelöl, amiket a program bármelyik részéről elérhetünk, beleértve azt az egységet is amiben definiáltuk.
- A `protected` (védett) kulcsszó olyan, részben védett metódusokat és mezőket takar, amiket csak az adott osztályból, és az abból származó alosztályokból lehet elérni.

Általában az osztály mezőinek – vagyis az adatoknak – célszerű private elérést beállítani, a metódusok – vagyis a kód – pedig lehet public. De ez sem mindig helyes, metódusok is lehetnek private vagy protected beállításúak, ha csak belső részfeladatok elvégzésére használatosak. Mezők is lehetnek public elérésűek, ha könnyű, közvetlen elérést szeretnénk biztosítani, és biztosak vagyunk abban, hogy az adattípusuk később sem változik.

Tipp

Public mezők helyett általában érdemes tulajdonságokat használni, amint azt a következő fejezetben láthatjuk. A tulajdonságok az információ elrejtési mechanizmus kiterjesztései az OOP nyelvekben, és nagyon fontosak az Object Pascalban.

Példaként álljon itt a TDate osztály új változata:

```
type
  TDate = class
  private
    Month, Day, Year: Integer;
  public
    procedure SetValue (m, d, y: Integer);
```

```
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  protected
    function DaysInMonth: Integer;
  end;
```

Ebben a változatban a mezők már private elérésűek, és van néhány új metódus is. Az első, a GetText függvény a dátumot szöveggé alakítva adja vissza. Ez az egyetlen módja, hogy a private adatokhoz hozzáférjünk, hacsak nem használunk tulajdonságokat. Esetleg felmerülhet, hogy szükség lenne más függvényekre, mint a GetDay (nap lekérdezés), GetMonth (hónap lekérdezés), vagy a GetYear (év lekérdezés), amik csak a megfelelő private adatot adják vissza, de az ilyen majdnem közvetlen adatelérést biztosító függvényekre nincs mindig igény.

A második új metódus az Increase (növel) eljárás, ami a dátumot megnöveli egy nappal. Ez az eljárás nem egyszerű, mert figyelni kell a hónapok hosszára, és a szökőévekre is.

Hogy leegyszerűsítsük ezt – és néhány később hozzáadandó új metódust –, írtunk egy függvényt, ami visszaadja a napok számát az aktuális hónapban, ez a DaysInMonth (HónapNapjai). Ez az eljárás nem része az osztály publikus interfészének, hiszen nincs túl sok értelme lekérdezni egy TDate objektumtól az adott hónap napjainak számát, ha azt sem tudjuk, hogy milyen hónap van. Két választásunk volt, a függvényt felvehettük akár private, akár protected elérésűre, ami miatt mégis protected lett, az az, hogy ha később örököltetni szeretnénk a TDate-ből egy új osztályt, akkor szükségünk lehet erre a funkcióra.

Public és published

A public, protected és private hozzáférési direktívák mellett használhatunk egy negyediket, az úgynevezett publishedet. A published mezők vagy metódusok nem csak futásidőben érhetők el, hanem tervezés alatt is. A Delphi komponenspalettáján lévő összes komponensnek van published interfésze, amit különböző Delphi eszközök használnak, különös tekintettel az Object Inspectorra. Ez az interfész a következő fejezetben bemutatásra kerülő összetettebb technikákkal érhető el.

A published interfész rendszeres használata nem az alkalmazás fejlesztésénél fontos, hanem komponensek írásakor. Általában a komponensek published részei nem tartalmaznak mezőket vagy metódusokat, csak egy új nyelvi elemet: a tulajdonságot (property). A tulajdonságokról a következő fejezetben lesz szó.

Amikor a Delphi formot készít, a saját metódusainak és komponenseinek definícióit a definíciók első részéhez teszi, még a public és private kulcsszavak elé. Ezek az osztálydefiníció elején szereplő mezők és metódusok publishedek. Egy osztály minden olyan eleménél, ahol nem adunk meg más kulcsszót, a published az alapértelmezett.

Célszerű publishednek felvenni az eseménykezelő metódusokat, és a formon lévő komponenseknek megfelelő mezőket, hogy automatikusan inicializálni lehessen azokat. Az Object Inspector által elérhető komponens, vagy metódus lis-tájában csak azok a komponensek illetve metódusok jelenhetnek meg, amiket a kezdeti, published részben deklaráltunk.

Erre végezhetünk is egy kísérletet. Adjuk hozzá a következő deklarációt a form első, published, részéhez:

```
procedure Foo (Sender: TObject);
```

Most válasszuk ki az Object Inspector Events oldalát, és nyissuk ki az OnCreate esemény melletti listát. Ebben szerepelni fog a Foo metódus, ha viszont ezt a definíciót kivágjuk, és átmásoljuk a public részre, a metódus nem fog megjelenni, mivel a Delphi nem készít belső leírást hozzá. Valójában akkor sem jelenne meg, ha egy új published részhez adnánk, mert a Delphi környezet csak a formdeklaráció elejét nézi át.

Osztályok és egységek

A Delphi alkalmazások, akár a legtöbb Pascal program, sok egységet, vagy program modult használnak. Az osztályok bevezetése előtt az egységek (unitok) voltak a modularitás alapjai a nyelvben, de, ahogy az a láthatósági szabályok magyarázatánál világosan kiderült, az osztályok is az egységek filozófiáját követik.

A Delphi alkalmazásokban minden form mögött van egy hozzá kapcsolódó egység, ha új formot adunk a projekthez (a megfelelő SpeedBar gombbal, vagy a **File** menü **New Form** parancsával), akkor a Delphi egy új egységet hoz létre, ami az új formot definiálja.

Minden formot egy egység határoz meg, de ez fordítva nem igaz, hiszen az egységeknek nem kell formokat definiálniuk; egyszerűen csak rutinok egy gyűjteményét tartalmazzák, és teszik elérhetővé. Ha a **File** menü **New** parancsának kiadása után az Object Repository **New** oldalán az Unit (egység) ikont kiválasztjuk új, üres egységet adhatunk az aktuális projekthez. Ez az üres egység a következő kódot tartalmazza, ami az egység szerkezeti elemeit mutatja:

```
unit Unit1;
interface
implementation
end.
```

Az egység koncepciója egyszerű. Minden egységnek van neve, ami projektenként egyedi, és az megegyezik egységet tartalmazó állomány nevével, egy interfész része, ahol a más egységek számára látható dolgokat definiálhatjuk, és egy kifejtés (implementation) rész, ahol a tényleges kód, és a többi, rejtett deklaráció található. Végül az egység tartalmazhat egy opcionális inicializáló részt, ahol az a kód kap helyet, ami az egység előkészítését végzi, és akkor fut le, amikor az egységet használó program betöltődik a memóriába, és egy másik opcionális rész, a befejezés, ami a program lezárásakor fut le. A befejező részt a Delphi 32 bites változatában vezették be, hogy az egység inicializáló részében lefoglalt dolgokat megfelelően felszabadíthassuk.

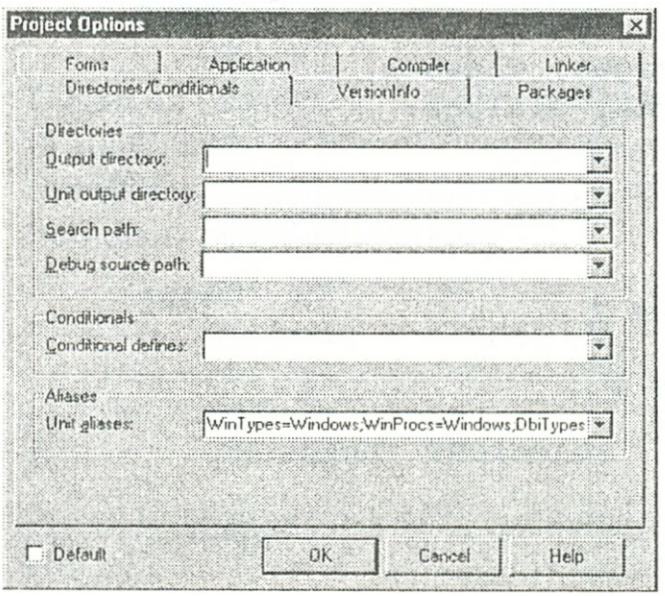
Az egység általános szerkezete, az összes lehetséges részt beleértve, a következő:

```
unit unitName;

interface

// other units we need to refer to
uses
  A, B, C;

// exported type definition
type
  newType = TypeDefinition;

// exported constants
const
  Zero = 0;

// global variables
var
  Total: Integer;

// list of exported functions and procedures
procedure MyProc;

implementation

uses
  D, E;

// hidden global variable
var
  PartialTotal: Integer;

// all the exported functions must be coded
```

```
procedure MyProc;
begin
  // ... code of procedure MyProc
end;

initialization
  // optional initialization part

finalization
  // optional clean-up code

end.
```

Az interface rész elején található uses kifejezésben szerepelnek azok az egységek, amikre az egység interface részében szükségünk lehet. Ez lehet olyan egység, amiben olyan adattípusok, komponensek vagy osztályok vannak, amikre más adattípusok, mint például az éppen definiált formon lévő komponensek meghatározásakor hivatkozunk.

Az implementation rész elején lévő második uses kifejezésben azok az egységek szerepelnek, amikre csak az implementation részben van szükség. Amikor egy másik egységre a rutinok és metódusok kódrészéből hivatkozunk, akkor azokat ehhez a második uses szekcióhoz adjuk, és ne az elsőhöz. Minden hivatkozott egységnek a projekt mappában, vagy egy másik, a keresőútvonalakon lévő mappában kell lennie. A projekt keresőútvonalait a **Project** menü **Options** parancsára megjelenő **Options** párbeszéblak **Directories/Conditionals** (könyvtárak és feltételes direktívák) oldalán adhatjuk meg, ahogy az az 5.6. ábrán látható.



5.6. ábra *A Project Options dialógus ablak Directories/Conditionals oldalán egységneveket, keresőútvonalakat és más opciókat állíthatunk be*

Megjegyzés

A C++ programozók figyelmébe ajánljuk, hogy a **uses** kifejezés nem felel meg az include direktívának. Az uses kifejezés csak arra szolgál, hogy a felsorolt egységek előfordított interfész részét importálja. Az egység implementációs része csak az egység fordításakor játszik szerepet.

A Delphi 32 bites verziójában jelent meg az új „unit aliasing" (unit átnevezés) technika, ez azt jelenti, hogy az uses kifejezésben egy egység nevét használjuk, de a fordító egy másik egységre hivatkozik. Ezeket a hivatkozásokat a **Project Options** dialógus **Directories / Conditionals** oldalán állíthatjuk be (lásd ismét az 5.6. ábrát).

Az egység interfészében több különböző elemet deklarálhatunk, ide tartoznak az eljárások, a függvények, a globális változók, és az adattípusok. Ez az utolsó fajta a leggyakrabban használt objektum orientált szempontból, és Delphi alkalmazásokban. Az osztályokat könnyen egységekbe helyezhetjük, és általában ez is a jó megoldás. A Delphi automatikusan ezt teszi, akárhányszor új formot készítünk. Emlékszünk még az új form kódjára, amit az első fejezetben néztünk?

```
unit Unit1;

interface

uses
  SysUtils, Windows, Messages, Classes,
  Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}
end.
```

Ebben az egységben az egyedüli exportált elem az új TForm1 adattípus definíciója, és az ebből az osztályból származó globális változó, a Form1. Észrevehetjük még a private és a public kulcsszavakat, és a $R fordító direktívával beszerkesztett form leíró állományt.

A Delphi egységeket nem csak formok tárolására használhatjuk, továbbra is lehetnek hagyományos egységeink, eljárásokkal és függvényekkel, és létrehozhatunk egységeket olyan osztályokkal, amik nem hivatkoznak formokra, vagy más vizuális elemekre.

Hogy megnézhessünk egy példát egy teljesen kidolgozott egységre, továbbfejlesztettük a TDate osztályt. Ha ezt az osztályt egy egységbe helyezzük, akkor, ahogy azt hamarosan látni fogjuk, bármilyen alkalmazás számára elérhetővé tehetjük. Azonban mielőtt ebbe belemerülnénk, nézzünk meg néhány osztály interfészekkel, láthatósággal és beágyazással kapcsolatos részletet az egységek esetében.

Osztály interfész

Amikor új osztályt hozunk létre, és egy egységbe helyezzük, az egység interfész részében megírjuk azt a részt, amit osztály interfésznek is hívnak (ez az osztály deklarációja). Az osztály interfész tartalmazza az adatok, és mezők deklarációját, valamint a metódusok előzetes deklarációját. Az adatdeklarációkra azért van szükség, hogy az osztályba tartozó objektumok méretét meg lehessen határozni, bár a private mezőket másik egységből nem lehet elérni.

Technikai megjegyzés, hogy az osztályt előzetesen deklarálhatjuk anélkül is, hogy a kifejtését azonnal elkészítenénk:

```
type
  MyClass = class;
```

Ugyan az osztályt nem tudjuk használni a teljes deklarációja nélkül, de ezt az új adattípust már használhatjuk más osztály mezőjeként, vagy más osztály metódusának paramétereként, ahogy azt a következő, két kapcsolódó osztályt bemutató példából láthatjuk:

```
type
  THusband = class;

  TWife = class
    Husband: THusband;
    ...
  end;

  THusband = class
    Wife: TWife;
    ...
  end;
```

Más OOP nyelvekben (mint például a C++), ez a kód hibás lenne, de Object Pascalban, az objektum-hivatkozásoknak köszönhetően teljesen jó. Elhelyezhetünk részlegesen definiált objektumokat – olyan osztályba tartozókat, amire csak előzetes hivatkozás van – másik osztályban, mert a fordító már tudja, hogy az objektumnak mennyi memóriára lesz szüksége, hiszen minden objektum annyi helyet foglal, amennyi egy hivatkozáshoz vagy egy mutatóhoz kell. Nem meglepő azonban, hogy C++-ban ilyen kódot csak explicit mutatók segítségével készíthetünk.

Egységek és érvényesség

Pascalban az érvényesség és a beágyazás az egységekre épül, ezért azok talán még az osztály private és public kulcsszavainál is fontosabbak, hiszen a private kulcsszó hatása is az osztályt tartalmazó egységre vonatkozik.

Egy azonosító – változó, eljárás, függvény, vagy adattípus – érvényessége alatt azt a kódrészt értjük, ahol az adott azonosító elérhető. Alapszabály, hogy egy azonosító csak az érvényességi területén értelmezhető, vagyis csak abban a blokkban, ahol deklaráltuk, ezen kívülről a változót nem lehet elérni. Íme néhány példa:

- Helyi, vagy lokális változók: ha egy változót egy rutin vagy metódus fejlécében deklarálunk, akkor azt azon az eljáráson kívülről nem lehet elérni. A változó érvényessége azonban kiterjed az egész eljárásra, beleértve a beágyazott rutinokat is, hacsak egy lentebbi szinten egy ugyanolyan nevű azonosító el nem takarja. A változó számára szükséges memóriaterület a deklaráló eljárás elindításakor foglalódik le a veremből, és az eljárás befejezésekor a veremterület automatikusan felszabadul.
- Globális rejtett változók: Ha egy azonosítót az egység implementation részében definiálunk, akkor azt az egységen kívül nem érhetjük el, de az egységen belül bármilyen eljárás vagy blokk használhatja. Az ilyen változók memóriaterülete a program indításától a lezárásáig foglalt, és az egység inicializáló részében kaphatnak kezdőértéket.
- Globális változók: Ha egy változót az egység interface részében deklarálunk, akkor az minden más olyan egységben is látható, ami használja a változót deklaráló egységet. Memória és élettartam tekintetében ezek a változók megegyeznek az előző csoporttal, az egyetlen különbség a láthatóságukban van.

Az egység interface részében elhelyezett minden deklaráció elérhető bármely olyan program minden részéből, ami az adott egységet az uses kifejezésben használja. A form változódeklarációk is ugyanezen a módón történnek, ezért egy formra – beleértve a public mezőit, metódusait, tulajdonságait és komponenseit is – bármely másik form kódjában hivatkozhatunk. Természetesen nem

túl jó programozási gyakorlat mindent globálisnak deklarálni: amellett, hogy így rengeteg memóriát pazarlunk, a globális változók használata a programot kevésbé teszi objektumorientálttá, és megnehezíti a forrás karbantartását és továbbfejlesztését. Célszerű a lehető legkevesebb globális változó és objektum használatára törekedni. A jövőben néhány példában láthatjuk, hogy néha még egy formot is célszerűbb lokálisként deklarálni, hogy megszabaduljunk a hozzá kapcsolódó globális változótól, és minél nagyobb a program, ez annál fontosabb lehet.

Egységek és érvényességi körök

Az uses kifejezés a bevett gyakorlat más egység változóinak elérésére. Innentől elérhetjük a másik egység definícióit is, de előfordulhat, hogy két hivatkozott egység tartalmaz egy ugyanolyan nevű azonosítót; vagyis van két osztályunk vagy rutinunk, amit ugyanúgy hívnak.

Ebben az esetben megtehetjük azt, hogy rutin vagy típus neve elé beírjuk az azt definiáló egység nevét. Például a Totals egységben lévő ComputeTotal eljárást írhatjuk Totals.ComputeTotal-nak. Erre általában nincs szükség, és nem is javasolt egy programban két különböző dolgot azonos névvel ellátni.

Ha a VCL könyvtárat, és a Windows állományokat nézzük, akkor azt találhatjuk, hogy néhány Delphi függvénynek ugyanaz a neve – bár a paraméterezése más -, mint néhány Windows API függvényé, amit a Delphiből is elérhetünk; erre példa a Beep eljárás.

Készítsünk egy új programot, tegyünk rá egy gombot, és írjuk be a következő kódot:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Beep;
end;
```

Amikor megnyomjuk a gombot, egy rövid hangjelzést hallhatunk. Most menjünk az egység uses kifejezéséhez, és az ott láthatókat erről:

```
uses
  Windows, Messages, SysUtils, Classes, ...
```

erre, a nagyon hasonló változatra módosítsuk, vagyis a SysUtils egységet tegyük a Windows egység elé:

```
uses
  SysUtils, Windows, Messages, Classes, ...
```

Ha most megpróbáljuk lefordítani a programot, a „Not enough actual parameters" (nincs elég paraméter) fordítási hibát kapjuk. A probléma oka az, hogy a Windows egység definiál egy másik Beep függvényt, két paraméterrel.

Általánosabban, az első egység definícióit elrejtheti egy másik, későbbi egységben lévő ugyanolyan nevű definíció. Erre a biztonságos megoldás a következő:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SysUtils.Beep;
end;
```

Ez a kód már az egységek sorrendjétől függetlenül lefordítható. Van még néhány másik névütközés a Delphiben, egyszerűen azért, mert a Delphi kódot általában osztályok eljárásai alkotják. Két különböző nevű osztály azonos metódusai nem okoznak problémát, a gondok csak a globális eljárásoknál jelentkezhetnek.

Változások elrejtése

Az adatelrejtés egyik legfontosabb alapelve a globális változók számának csökkentése. Mivel a globális változókat az egész programból el lehet érni, így az ezeket érintő változások kihatnak az egész programra. Ezzel szemben, ha egy osztály egy mezőjét változtatjuk meg, akkor elég csak annak az osztálynak néhány metódusát átírni, és semmi mást. Emiatt azt mondhatjuk, hogy az információ elrejtés elrejti a változásokat is.

Hogy érthetőbbé tegyük ezt, nézzünk egy példát. A LeapYear példában a TDate objektumot az egység interfész részének változói között deklaráltunk. Ezt az objektumot az egység bármely részéből elérhetjük, de akár az alkalmazás bármely másik egységből is látható, hiszen az interfész részben deklaráltuk. A mi kis programunkban ez rendben is volt, de egy összetettebb alkalmazásban, amit több programozó fejleszt, más egységek is használhatják ezt az objektumot, ebben az esetben pedig a TDate objektum nevének, vagy használati módjának megváltoztatása az egész programot tönkreteheti.

Hogy ezt elkerüljük, a TDate osztályt felvehettük volna a formot leíró osztály mezőjeként:

```
type
  // new data type, a TDate
  TDate = class
  private
    Month, Day, Year: Integer;
  public
    constructor Init (m, d, y: Integer);
    function LeapYear: Boolean;
  end;

  TForm1 = class(TForm)
```

```
    Btn1997: TButton;
    Btn1996: TButton;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Button1995Click(Sender: TObject);
    procedure Button1996Click(Sender: TObject);
  private
    { Private declarations }
    ADay: TDate; // new private declaration!
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

Ha ezt private mezőként deklaráljuk, akkor a TDate objektumot nem lehet elérni az egységen kívülről, hacsak nem biztosítunk erre egy megfelelő metódust. Ha viszont írunk egy elérést szolgáló metódust, és később úgy döntünk, hogy megváltoztatjuk az objektum jelentését vagy megjelenési formáját, akkor csak az elérést szolgáló metódust kell ennek megfelelően módosítani, és ez nem hat ki a program többi részére.

A TDate osztály három mezőjének private deklarációja elősegíti az információ elrejtést, és ettől a programunk objektumorientáltabb lesz, de, hogy ez tényleg jól működjön, ahhoz írnunk kell az osztálynak néhány metódust, amikkel elérhetjük a private mezőket.

Megjegyzés

Van még egy második különbség is aközött, ha mezőt adunk a form osztályhoz, és ha globális változót deklarálunk az egységben. Az első esetben, ha két formot készítünk ugyanabból az osztályból, akkor az adatok két példányban fognak létezni, míg az egység adatait a form osztály minden objektuma megosztva fogja használni. Természetesen, ennek csak másodlagos formok esetében van értelme, amikből lehet, hogy több példányt akarunk készíteni, hiszen az alkalmazások főablakai általában egyediek.

Egység a TDate osztály számára

Ebben a fejezetben sok példát készítettünk a TDate osztály egyszerű változataival. Az osztály végleges változatának több funkciója lesz, és egy egységben fogjuk azt elhelyezni. A következő listán az osztály interfésze látható, ami az egység interfészének része.

```
unit Dates;

interface

type
  TDate = class
  private
    Month, Day, Year: Integer;
  public
    constructor Init (m, d, y: Integer);
    procedure SetValue (m, d, y: Integer);
    function LeapYear: Boolean;
    procedure Increase;
    procedure Decrease;
    procedure Add (NumberOfDays: Integer);
    procedure Subtract (NumberOfDays: Integer);
    function GetText: string;
  protected
    function DaysInMonth: Integer;
  end;
```

Az új metódusok hatását elég egyszerű megérteni:

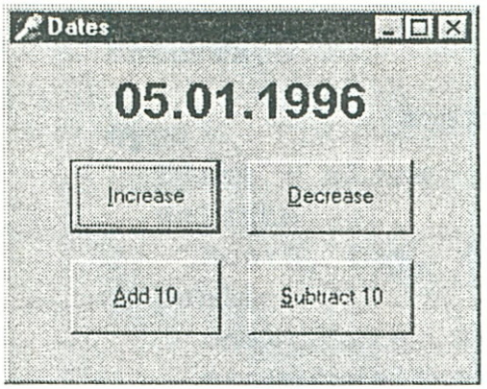
- Az Increase (növel) és Decrease (csökkent) a dátumot változtatják meg egy nappal előre vagy vissza.
- Az Add (hozzáad) és Subtract (kivon) eljárások a dátumot a paraméterként megadott napszámmal növelik vagy csökkentik. Például ha az objektum értéke 1997-03-08, akkor 10 nap hozzáadásával ez 1997-03-18-ra változik.
- A GetText (szöveglekérdezés) a dátumot formázott stringként adja vissza, amit a Format (formázás) függvény és néhány formázási szabály segítségével állítunk elő.

A következő elég hosszú listán az osztály néhány metódusának forrása látható. Az egész kódot a CD-n az egység kipróbálására készült ViewDate példa részeként találhatjuk meg.

```
function TDate.DaysInMonth: Integer;
begin
  case Month of
```

```
    1, 3, 5, 7, 8, 10, 12:
      DaysInMonth := 31;
    4, 6, 9, 11:
      DaysInMonth := 30;
    2:
      if (LeapYear) then
        DaysInMonth := 29
      else
        DaysInMonth := 28;
    else
      // if the month is not correct
      DaysInMonth := 0;
  end;
end;

procedure TDate.Increase;
begin
  // if this day is not the last of the month
  if (Day < DaysInMonth) then
    Inc (Day) // increase the value by 1
  else
  // if it is not in December
    if Month < 12 then
    begin
      // Day 1 of next month
      Inc (Month);
      Day := 1;
    end
    else
    begin
      // else it is next year New Year's Day
      Inc (Year);
      Month := 1;
      Day := 1;
    end;
end;

function TDate.GetText: string;
begin
  {format the text, converting the integers to
  strings with a fixed number of characters}
  GetText := Format ('%.2d.%.2d.%4d',
    [Month, Day, Year]);
end;

procedure TDate.Add (NumberOfDays: Integer);
var
```

```
    N: Integer;
  begin
    // increase the day n times
    for N := 1 to NumberOfDays do
      Increase;
  end;
```



5.7. ábra *A ViewDate példa elindítás után*

Az egység kipróbálására készíthetünk egy másik példát. Az új formon lesz egy címke, ami a dátumot mutatja, és négy gomb, amikkel ezt a dátumot módosíthatjuk. A ViewDate példa főablakának képét az 5.7. ábrán láthatjuk. Figyeljük meg, hogy a jó működés érdekében a címke nagyméretű, középre igazított, olyan széles, mint form, és az AutoSize (automatikus méretezés) tulajdonságát hamisra állítottuk, ahogy az a form szöveges leírásából látható:

```
object DateForm: TDateForm
  Caption = 'Dates'
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OnCreate = FormCreate
  OnDestroy = FormDestroy
  object LabelDate: TLabel
    Alignment = taCenter
    AutoSize = False
    Caption = 'date'
    Font.Height = -27
    Font.Name = 'Arial'
    Font.Style = [fsBold]
  end
  object BtnIncrease: TButton
    Caption = '&Increase'
    OnClick = BtnIncreaseClick
```

```
    end
    object BtnDecrease: TButton
      Caption = '&Decrease'
      OnClick = BtnDecreaseClick
    end
    object BtnAdd10: TButton
      Caption = '&Add 10'
      OnClick = BtnAdd10Click
    end
    object BtnSubtract10: TButton
      Caption = '&Subtract 10'
      OnClick = BtnSubtract10Click
    end
  end
```

A program induló kódját az OnCreate eseményben készíthetjük el. Az ennek megfelelő metódusban elkészítjük a TDate osztály egy példányát, kezdőértéket adunk neki, és a címkét feltöltjük a szöveges leírásával, ahogy az az 5.7. ábrán láthatjuk:

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TDate.Init(5, 1, 1997);
  LabelDate.Caption := TheDay.GetText;
end;
```

A TheDay most már a TdateForm adattípus private mezője. Itt jegyezzük meg, hogy az osztály nevét, a TDateForm-ot, a Delphi automatikusan választotta, miután a form Name tulajdonságát DateForm-ra változtattuk.

Amikor a négy gomb egyikét megnyomjuk, meg kell hívnunk a TheDay objektum megfelelő metódusát, és be kell állítanunk a címke új értéket. Íme két példa:

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);
begin
  TheDay.Increase;
  LabelDate.Caption := TheDay.GetText;
end;

procedure TDateForm.BtnAdd10Click(Sender: TObject);
begin
  TheDay.Add(10);
  LabelDate.Caption := TheDay.GetText;
end;
```

Tipp

Ahogy az az 5.7. ábrán is látszik, a form négy gombjának az első betűje alá van húzva, ami azt jelenti, hogy ezeket a karaktereket gyorsbillentyűkként használhatjuk. Ehhez elég, ha a gomb Caption tulajdonságában az egyik betű elé teszünk egy & karaktert, ahogy azt a korábban látott gombok esetében tettük. A gyorsbillentyű használatával könnyebben változtathatjuk a dátumot, mert ha a négy gomb egyikét lenyomva tartjuk, akkor a dátum gyorsan fog változni.

Ha még mindig nem világos a program működése, akkor a CD-n megtalálhatjuk a teljes forrást további vizsgálatra, és módosításra. Például megpróbálhatunk olyan gombokat készíteni, ami több nappal változtatja a dátumot, vagy készítsünk olyan Következő Év gombot, ami szökőévek esetén is helyesen működik.

Egységek és programok

Egy Delphi alkalmazás kétféle forrásállományból épül fel: egy vagy több egységből, és egy program állományból. Az egységeket olyan másodlagos állományoknak tekinthetjük, amikre az alkalmazás fő része, a program hivatkozik. Ez azonban csak elméletben igaz; gyakorlatban a program állomány általában egy korlátozott szereppel bíró automatikusan generált fájl, amire csak annyi szükség van, hogy elinduljon a program, és megjelenjen a főform. Ahogy azt már az első és második fejezetben láthattuk, a program állomány – vagy Delphi projekt állomány (.dpr) – szerkeszthető akár kézzel, akár a Project Manager, vagy néhány, az Application objektummal és a formokkal összefüggő projekt opció segítségével.

A program állományszerkezete általában sokkal egyszerűbb, mint az egységeké, példaként álljon itt a ViewDate alkalmazás program állományának forrása:

```
program ViewDate;

uses
  Forms,
  DateF in 'DATEF.PAS' {DateForm},
  Dates in 'DATES.PAS';

begin
  Application.Initialize;
  Application.CreateForm(TDateForm, DateForm);
  Application.Run;
end.
```

Amint látható, itt csak egy `uses` kifejezés van és az alkalmazás `begin` és `end` kulcsszavak közé zárt főkódja. A program uses kifejezése különösen fontos, hiszen ez irányítja az alkalmazás fordítását és szerkesztését.

Megjegyzés

A Delphi 32 bites változatában a project kódjában eggyel több sor van, mint a 16 bites változatban. Az új sor, az Application.Initialize általában nem csinál semmit, és nem is szükséges. Erre csak akkor van szükség, ha a projektben van a TAutoObject-nek – ami az alkalmazást OLE Automation szerverré teszi – alosztálya, ebben az esetben ugyanis az Initalization metódus aktualizálja a rendszer regisztrációs adatbázisát. Erről bővebben a 22. fejezetben lesz szó.

Öröklés meglévő típusokból

Gyakran előfordul, hogy egy létező osztály némileg módosított változatára van szükségünk. Lehet, hogy egy új metódust kell hozzáadnunk, vagy egy már létezőt kell átírnunk, amit könnyen megtehetünk az eredeti forrás átírásával, hacsak nem használjuk ugyanezt az osztályt a programunk egy másik egységében. Ebben az esetben ugyanis a változások gondokat okozhatnak a másik egységben, emiatt ha tehetjük, kerüljük el a más egységeket érintő módosításokat, különösen nagy alkalmazások esetében.

Tipikus megoldás, hogy lemásoljuk az eredeti típusdefiníciót, módosítjuk a kódját, és új nevet adunk a végeredménynek. Ez működik, de felvet problémákat is: a kód másolásakor lemásoltuk a hibákat is, és ha új funkciót akarunk hozzáadni, akkor az a másolatok számától függően két vagy több példányban kell megtennünk. Ennek az eredménye két teljesen különböző adattípus lesz, és a fordító nem tudja kihasználni a típusok közötti hasonlóságokat.

Az osztályok hasonlóságát illető problémák megoldására az Object Pascal lehetővé teszi, hogy egy osztályt közvetlenül, egy már létező másik osztályból hozzunk létre. Ezt a technikát hívják öröklésnek, és egyike az objektumorientált nyelvek alapelemeinek. Ahhoz, hogy egy létező osztályból örököltessünk, az alosztály deklarációjának elején jeleznünk kell a szülő osztályt. Például a Delphi automatikusan így tesz mindig, amikor új formot készítünk:

```
type
  TForm1 = class(TForm)
  end;
```

Ez az egyszerű definíció azt jelenti, hogy a TForm1 osztály a TForm osztály összes metódusát, mezőjét és egyéb jellemzőjét örökli. Egy TForm1 osztályú objektumra a TForm osztály akármelyik public metódusát alkalmazhatjuk. Ha megnézzük a Súgóban a TFormot, akkor láthatjuk, hogy sok metódusa van. A TForm viszont örököl néhány metódust egy másik osztályból, és így tovább, egészen a TObject osztályig (a TObject osztályról ismertető található a következő fejezetben).

Megjegyzés

A Delphi 32 bites változataiban egy form lehet egy másik saját készítésű form alosztálya. Ezt a hasznos technikát vizuális form öröklésnek hívják, és a 13. fejezetben lesz szó róla, miután átnéztünk több formos programot.

Az öröklésre egyszerű példaként a ViewDate programot átírhatjuk egy kicsit, úgy, hogy egy új osztályt hozunk létre a TDate osztályból, aminek GetText függvényét módosítjuk. Ezt a forrást a kísérő lemezen a ViewD2 példa DATES.PAS állományában találhatjuk.

```
type
  TNewDate = class (TDate)
  public
     function GetText: string;
  end;
```

Ebben a példában TNewDate a TDate osztályból származik. Ezt gyakran úgy mondjuk, hogy a TDate a TNewDate ős, vagy szülő osztálya (vagy bázis osztálya, a C++ terminológiában), és hogy a TNewDate a TDate gyermek, al-, vagy leszármazott osztálya (vagy származtatott osztálya, C++-ban).

Figyelem!

Különböző objektum-orientált nyelvek különböző kifejezéseket használnak az öröklésre, és az öröklésben résztvevő osztályokra. Sajnos nincs egységes szóhasználat a nyelvek, de még a programozók között sem.

A GetText függvény első új verziójában a dátum leírásához a hónapok nevét egy konstans tömbben tároljuk, amit az egység implementation részében definiálunk:

```
// definition of the month names
const
  MonthNames: array [1..12] of string =
    ('January', 'February', 'March', 'April',
    'May', 'June', 'July', 'August', 'September',
    'October', 'November', 'December');

function TNewDate.GetText: string;
begin
  GetText := Format ('%s %d, %d',
    [MonthNames[Month], Day, Year]);
end;
```

Később rájöttünk, hogy sokkal jobb a Windowsban elérhető előre definiált hónapneveket használni, hiszen ezek a helyi beállításokat tükrözik. Ezeknek a helyi beállításoknak a legnagyobb részét a Delphi a könyvtárban definiált konstansokban átveszi, mint amilyen a LongMonthNames (hosszú hónapnevek), ShortMonthNames (rövid hónapnevek), és még sok más, amit a Delphi súgóban a „Currency and date/time formatting variables" – pénz-, dátum-, és időformátum változók – címszó alatt találhatunk meg. Itt a GetText metódus egy jobb változata:

```
function TNewDate.GetText: string;
begin
  GetText :=  Format ('%s %d, %d',
    [LongMonthNames[Month], Day, Year]);
end;
```

Tipp

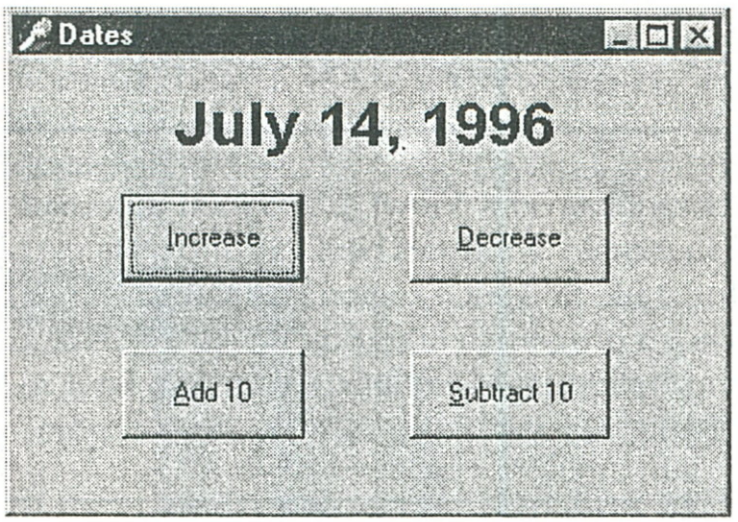
A regionális beállítások használatával a ViewD2 program automatikusan alkalmazkodik a felhasználó Windows beállításaihoz. Ha ezt a programot olyan gépen futtatjuk, aminek a regionális beállításai nem angol nyelvre utalnak, akkor a program a hónapok neveit a beállított nyelven jeleníti meg. Ahhoz hogy ezt kipróbálhassuk, elég, ha a regionális beállításokat átállítjuk, nincs szükség új Windows verzióra.

Ez a kód csak akkor működik, ha ugyanabba az egységbe írjuk, amibe a TDate osztályt, hiszen az ős osztály private mezőire hivatkozunk. Ha a leszármazott osztályt másik egységbe szeretnénk helyezni, akkor a három mezőt vagy protectedként kellene deklarálnunk, vagy biztosítanunk kellene három egyszerű metódust az ős osztályban a három private metódus olvasására.

Miután létrehoztuk az új osztályt, felhasználhatjuk a ViewD2 példa forrásában. Egyszerűen definiáljuk a TheDay objektumot TNewDate típusúnak, és a FormCreate metódusban hívjuk meg a konstruktorát:

```
type
  TDateForm = class(TForm)
    ...
  private
    TheDay: TNewDate; // updated declaration
  end;

procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := NewDate.Init (7, 4, 1997); // updated
  DateLabel.Caption := TheDay.GetText;
end;
```

A ViewD2 példa minden további változtatás nélkül tökéletesen működik, hiszen a TNewDate osztály örökölte a dátum növelésére, napok hozzáadására szolgáló metódust csakúgy, mint az összes többit, tehát, az ezeket hívó régi kód még mindig működik. Sőt, még az új GetText metódus meghívásához sem kell megváltoztatnunk a forrást! A Delphi fordító azt a hívást az új metódusra irányítja. A többi eseménykezelő forrása is a régi marad, bár a jelentésük megváltozik, ahogy azt az új végeredmény mutatja (5.8. ábra).



5.8. ábra *A ViewD2 program kimenete, regionális beállításoktól függő hónapnevekkel*

Öröklés és típus kompatibilitás

A Pascal egy szigorúan típusos nyelv, ami azt jelenti, hogy egy logikai változó nem vehet fel egész értéket, legalábbis explicit típusválasztás nélkül, hiszen ez értelmetlen lenne. A szabály az, hogy két érték csak akkor kompatíbilis, ha ugyanolyan típusúak, vagy, precízebben, ha az adattípusuknak ugyanaz a neve.

Osztálytípusoknál azonban van egy fontos kivétel. Ha például készítünk egy TAnimal osztályt, és ebből származtatunk, mondjuk, egy TDog osztályt, akkor egy TAnimal típusú változónak értékül adhatunk egy TDog típusú objektumot, mégpedig azért, mert a kutya az egy állat. Lehet, hogy meglepő, de az alábbi konstruktor hívások mind jók:

```
var
  MyAnimal, MyAnimal2: TAnimal;
begin
  MyAnimal := TAnimal.Create;
  MyAnimal2 := TDog.Create;
  ...
```

Általános szabályként azt mondhatjuk, hogy leszármazott osztálybeli objektumot minden olyan esetben használhatunk, amikor ős osztálybeli objektumra van szükség. Ennek a fordítottja azonban nem igaz, gyermek objektum helyett nem használhatjuk az ős objektumot. Kód formájában (az első sor jó, a második hibás):

```
MyAnimal := MyDog;   // This is OK
MyDog := MyAnimal;   // This is an error!!!
```

Mielőtt belemennénk ennek a fontos tulajdonságnak a megvalósításába, nézzünk egy példát, az Animals1-et, amit később még kibővítünk. A példa két osztályát a következőképp definiáltuk az ANIM.PAS egységben:

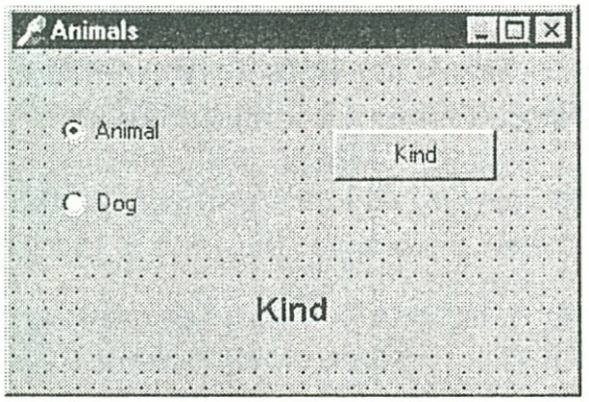
```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
  private
    Kind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
  end;
```

A két Create metódus egyszerűen csak beállítja a fajt, amit a GetKind függvény visszaad:

```
constructor TAnimal.Create;
begin
  Kind := 'An animal';
end;

function TAnimal.GetKind: string;
begin
  GetKind := Kind;
end;

constructor TDog.Create;
begin
  Kind := 'A dog';
end;
```

Hogy bemutassuk az osztályok használatát, készítettünk egy egyszerű formot két rádiógombbal, egy gombbal, és egy címkével, aminek nagy, középre igazított felirata van, ahogy az az 5.9. ábrán és a következő listarészleten is látszik:

```
object FormAnimals: TFormAnimals
  Caption = 'Animals'
  OnCreate = FormCreate
  OnDestroy = FormDestroy
  object LabelKind: TLabel
    Alignment = taCenter
    AutoSize = False
    Caption = 'Kind'
    Font.Height = -16
    Font.Name = 'Arial'
    Font.Style = [fsBold]
  end
  object BtnKind: TButton
    Caption = '&Kind'
    OnClick = BtnKindClick
  end
  object RbtnAnimal: TRadioButton
    Caption = '&Animal'
    Checked = True
    OnClick = RbtnAnimalClick
  end
  object RbtnDog: TRadioButton
    Caption = '&Dog'
    OnClick = RbtnDogClick
  end
end
```



5.9. ábra *Az Animals1 példa formja tervezés alatt*

A formnak van néhány mezője, ami állatok értékeit tárolja, egy kutyáét, és egy általános objektumét, a MyAnimal-ét:

```
type
  TFormAnimals = class(TForm)
    ...
  private
    MyAnimal, AnAnimal: TAnimal;
    ADog: TDog;
  end;
```

Az objektumok egyedeit a form készítésekor létrehozzuk és inicializáljuk:

```
procedure TFormAnimals.FormCreate(Sender: TObject);
begin
  AnAnimal := TAnimal.Create;
  ADog := TDog.Create;
  MyAnimal := AnAnimal;
end;
```

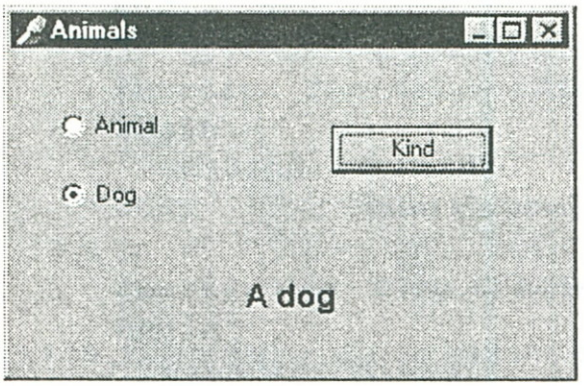
A két rádiógomb OnClick eseménykezelője szolgál arra, hogy a MyAnimal változóhoz kapcsolt objektumot kicserélje. Itt az egyik:

```
procedure TFormAnimals.RbtnDogClick(Sender: TObject);
begin
  MyAnimal := ADog;
end;
```

Végül a Kind gomb meghívja az aktuális állat GetKind metódusát, és az eredményt megjeleníti a címkén.

```
procedure TFormAnimals.BtnKindClick(Sender: TObject);
begin
  KindLabel.Caption := MyAnimal.GetKind;
end;
```

A metódus végeredménye az 5.10. ábrán látható. Ne felejtsük el a FormDestroy metódusban megsemmisíteni a két objektumot (az ADog-ot és az AnAnimal-t). Ha ez a kompatibilitási szabály világos, akkor elkezdhetjük kihasználni az előnyeit, a késői kötés tárgyalásával.



5.10. ábra *Az Animals1 példa a Dog rádiógomb kiválasztása után*

Késői kötés és polimorfizmus

A Pascal függvények és eljárások általában a statikus vagy korai kötésen alapulnak. Ez azt jelenti, hogy a metódushívást a fordító vagy a szerkesztő oldja fel, felcserélve azt egy, arra a memóriacímre utaló hívással, ahol a függvény vagy eljárás található (ezt a függvény címének is hívják).

Az objektumorientált programozási nyelvek, mint amilyen az Object Pascal is, egy másfajta kötési formát is támogatnak, az úgynevezett dinamikus, vagy késői kötést. Ebben az esetben a meghívott metódus valós címe csak futásidőben derül ki. Ennek a technikának az előnye polimorfizmus néven ismert. A polimorfizmus alapötlete az, hogy megírunk egy metódushívást, ezt alkalmazzuk egy változón, de az, hogy valójában melyik metódust hívjuk meg, az a változó által mutatott objektum típusától függ. A fenti típus-kompatibilitási szabály miatt a Delphi csak futás közben tudja eldönteni a változó által mutatott objektum osztályát.

Megjegyzés

A polimorfizmus elég bonyolult szó, ha megnézzük a szótárban, azt találhatjuk, hogy valami, aminek több alakja van. OOP-s értelemben ez azt a tényt jelenti, hogy egy metódusnak több változata lehet, és egy metódushívással ezek bármelyikét meghívhatjuk.

Például vegyük azt, hogy egy osztály és az alosztálya – mondjuk a TAnimal, és a TDog – is definiál egy metódust, és erre a metódusra késői kötés érvényes. Most ezt a metódust alkalmazhatjuk egy általános változóra, mint amilyen a MyAnimal, ami futásidőben akár TAnimal, akár TDog osztályú objektumra mutathat. Az, hogy melyik legyen ténylegesen meghívott metódus, futásidőben dől el, az aktuális objektum osztályától függően.

Az Animals2 példa kibővíti az Animals1-et, hogy bemutassa ezt az új technikát. A program új változatában a TAnimal és a TDog osztályoknak van egy új metódusa, a Verse – ez a kiválasztott állat hangját játssza le, és írja ki. Ezt a metódust a TAnimal osztályban virtuálisnak definiáltuk, és később a TDog osztályban felülírtunk a virtual és az override (felülbírál) kulcsszavak segítségével.

```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
    function Verse: string; virtual;
  private
    Kind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
    function Verse: string; override;
  end;
```

Természetesen a két metódust ki is kell fejteni, valahogy így:

```
uses
  MMSystem;

function TAnimal.Verse: string;
begin
  Verse := 'Verse of the animal';
  PlaySound ('Anim.wav', snd_Async, 0);
end;

function TDog.Verse: string;
begin
  Verse := 'Arf Arf';
  PlaySound ('dog.wav', snd_Async, 0);
end;
```

Megjegyzés

Ez a példa az MMSystem egységben definiált PlaySound API függvényt használja. A függvény első paramétere a lejátszani kívánt hangállomány neve, a második és a harmadik pedig a függvény viselkedését állítja. Erről az egyszerű multimédia alkalmazások tárgyalásánál lesz szó, a 31. fejezetben.

Akkor mi a hatása a MyAnimal.Verse hívásnak? Az attól függ. Ha a MyAnimal változó éppen egy TAnimal osztályú objektumra mutat, akkor a TAnimal.Verse – ez a rövidebb forma jelzi a TAnimal osztály Verse metódusát – metódust hívja, ám ha a MyAnimal objektum egy TDog osztályú objektumra mutat, akkor a hívás egy TDog.Verse lesz. Ez csak azért történhet, mert a függvény virtuális, és ezért késői kötésre alkalmas.

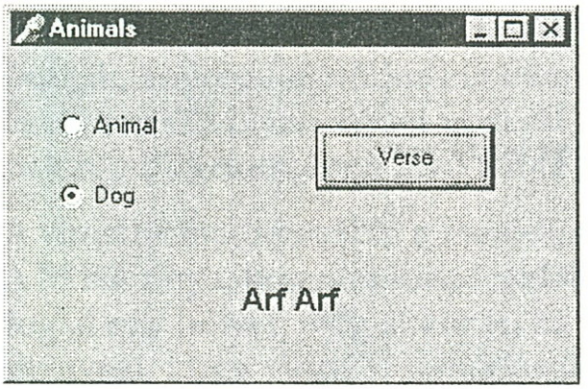
A MyAnimal.Verse hívás a TAnimal osztály minden leszármazottján értelmezett, még akkor is, ha azokat az osztályokat ennek a hívásnak az érvényességén kívül definiáltuk. A fordítónak nem kell ismernie az összes leszármazottat ahhoz, hogy a hívást érvényessé tegye, elég az ősosztályt. Más szóval a MyAnimal.Verse hívás kompatíbilis az összes még el nem készített TAnimal alosztállyal.

Ez az egyik legfontosabb technikai ok, ami ahhoz vezetett, hogy az objektumorientált nyelvek támogatják az újrafelhasználást. Írhatunk kódot, ami egy hierarchia osztályait használja, anélkül, hogy bármit is tudnánk a hierarchiáról. Más szóval, a hierarchiát – és a programot – még akkor is kibővíthetjük, amikor már több ezer kódsor használja. Természetesen ennek van egy feltétele: a hierarchiák ősosztályait nagyon gondosan meg kell tervezni.

Az Animals2 program formja hasonló az előzőhöz, de ez az új osztályok használatát mutatja be. A gomb felirata most Verse, és új a Name tulajdonsága is (BtnVerse). A címke Name tulajdonsága LabelVerse lett. Ez a kód fut le, ha a gombra kattintunk:

```
procedure TFormAnimals.BtnVerseClick(Sender: TObject);
begin
  LabelVerse.Caption := MyAnimal.Verse;
end;
```

Az 5.11. ábrán a program egy lehetséges eredményét láthatjuk. Futtatáskor a PlaySound (HangLejátszás) API hívás által keltett hangot is hallhatjuk.



5.11. ábra *Az Animals2 példa kimenete*

Ha összehasonlítjuk a két programot, arra gondolhatunk, hogy „mi is van most?". Bár a két program viselkedése és megjelenése nagyon hasonló, a színfalak mögött egészen más történik. Az Animals1-ben egy függvényhívás volt, a GetKind, ami egyszerűen visszaadta egy mező értékét. Az Animals2-ben viszont csak egy hívás van, a MyAnimal.Verse, ami néha az egyik, néha a másik osztály egy metódusát jelenti. Ez egy érdekes, és hasznos technika, hiszen ugyanaz a lefordított kód hívja meg valamelyiket a függvények közül az objektum típusától függően, és ez futásidőben dől el.

Hogy jobban megérthessük, miért fontos ez, tegyük föl, hogy a két vagy több metódus nagyon összetett, és teljesen különböző is lehet. Virtuális metódusokat hívhatunk az osztály másik metódusából, és virtuális metódusok hívhatnak más virtuális és statikus metódusokat is.

Metódusok felülírása és újradefiniálása

Ahogy azt láthattuk, egy késői kötéses metódus felülírásakor a leszármazott osztályban az `override` kulcsszót kellett használnunk, ezt azonban csak akkor tehettük meg, ha ezt a metódust az ősben virtuálisnak (virtual) definiáltuk, másképp ez statikus metódus lett volna, ahol pedig az ős megváltoztatása nélkül nincs mód a késői kötésre.

A szabályok egyszerűek: egy statikusnak definiált osztály minden alosztályban is statikus marad, egészen addig, ameddig el nem fedjük egy ugyanilyen nevű virtuális metódussal. A virtuálisnak definiált metódusok azonban késői kötésűek maradnak minden alosztályban is, ezt nem lehet megváltoztatni, mert a fordító a késői kötésű metódusoknak másfajta kódot generál.

Egy statikus metódus újradefiniálásakor elég, ha létrehozunk az alosztályban egy ugyanilyen nevű metódust, aminek a paraméterezése megegyezhet az eredetiével, de el is térhet attól. Virtuális metódus felülírásakor a paraméter listának meg kell egyeznie, és használnunk kell az `override` kulcsszót:

```
type
  MyClass = class
    procedure One; virtual;
    procedure Two; {static method}
  end;

  MySubClass = class (MyClass)
    procedure One; override;
    procedure Two;
  end;
```

Egy metódus felülírására két különböző lehetőség van. Az egyik, hogy az ős osztály metódusát egy újra cseréljük, a másik pedig, hogy a már létező metódushoz adjuk hozzá a kódot. Ez utóbbit az inherited (örökölt) kulcsszóval tehetjük meg, ami az ős osztály ugyanilyen metódusát hívja. Például megtehetjük ezt:

```
procedure MySubClass.One;
begin
  // new code
  ...
  // call inherited procedure MyClass.One
  inherited One;
end;
```

Talán felmerült, hogy miért van szükség az `override` kulcsszóra, hiszen más nyelvekben, ha egy alosztály egy metódusát újradefiniáljuk, akkor az automatikusan felülírja az eredetit. Ez a kulcsszó azonban lehetővé teszi, hogy a fordító leellenőrizhesse az összefüggést az ős és a leszármazott metódusnevei között, azzal, hogy megnézi, hogy az adott metódus az ősben virtuális volt-e, és így tovább. Más OOP nyelvekben gyakori hiba volt az újradefiniált függvények neveinek elgépelése.

Továbbá, ha egy könyvtár bármely osztályából örököltetett osztályban felveszünk egy statikus metódust, akkor még olyankor sem jelentkezik probléma, ha az eredeti könyvtárban ezt a nevet egy virtuális metódus kapja. Mivel a metódusunknál nem szerepel az override kulcsszó, ezért külön eljárás lesz, és nem a könyvtárban lévő egy újabb verziója, ami lehet, hogy elrontaná a programunkat.

Virtuális és dinamikus metódusok

A Delphiben két módja van annak, hogy egy metóduson késői kötést alkalmazzunk: virtuálisnak (`virtual`), vagy dinamikusnak (`dynamic`) kell azt deklarálnunk. A két kulcsszó szintaxisa és hatása teljesen megegyezik, a különbség a fordító által a késői kötés megvalósítására alkalmazott technikában van.

A virtuális metódusok egy virtuális metódus táblán – VMT, vagy vtable – alapulnak, ami egy metódus címeket tartalmazó tömb. Virtuális metódus hívásakor a fordító olyan kódot generál, ami az objektum virtuális metódus táblájának n-edik rekeszében lévő címre ugrik.

A virtuális metódustábla a metódushívások gyors lebonyolítását teszi lehetővé. A fő hátránya, hogy minden leszármazott osztály minden virtuális metódusa elfoglal egy helyet, még akkor is, ha az alosztályban nem írtuk felül. Ez azt eredményezi, hogy virtuális metódustábla bejegyzéseket vezetünk át az osztályhierarchián, még olyan eljárásokét is, amiken nem változtattunk. Az azonos metódusok címeinek sokszori letárolása pedig rengeteg memóriát foglalhat.

A dinamikus metódushívások egy, a metódusra jellemző, egyedi érték alapján működnek. A megfelelő funkció megkeresése pedig általában lassabb, mint egy egyszerű táblaelem kivétel a virtuális metódusoknál. A dinamikus metódusok előnye viszont, hogy csak akkor viszik tovább a metódus bejegyzést a leszármazottba, ha az felülírja az adott metódust. Nagyobb, vagy mélyebb hierarchiák esetén a dinamikus metódusok használata jelentős memória megtakarítást eredményezhet minimális sebességcsökkenés árán.

Programozói szempontból a két megoldás között különbség csak az eltérő belső ábrázolásban, a kissé eltérő sebességben, és a memóriaigényben van. Ezen kívül a virtuális és dinamikus metódusok megegyeznek.

Néhány irányadó szabály:

- Ha egy metódust szinte minden leszármazottban felülírunk, akkor azt virtuálisnak vegyük fel.
- Ha a metódust nem írjuk felül túl gyakran, de a késői kötésre szükség van a rugalmasság miatt, akkor ezt dinamikusnak deklaráljuk, különösen akkor, ha ennek az osztálynak sok leszármazott osztálya lesz.
- Ha a metódust másodpercenként sokszor meg kell hívni, akkor mindenképpen virtuális legyen, egyéb esetekben nincs valódi sebességkülönbség a két fajta között.

Természetesen ez azt feltételezi, hogy már eldöntöttük, melyik metódusoknak kell késői kötést használniuk. A statikus metódusok a metódushívás leggyorsabb változatai, ezért ez az alapértelmezett.

Üzenetkezelők

Késői kötésű metódusokat Windows üzenetek kezelésére is használhatunk, bár ez a technika némiképp eltérő az előbbiektől. Erre a célra a Delphi egy másik direktívát használ, ez a `message` (üzenet), ami üzenetkezelő metódust definiál. Az ilyen metódusoknak egyetlen var paraméterrel rendelkező

eljárásoknak kell lenniük. Ezt követi a message direktíva és kezelni kívánt Windows üzenet sorszáma, vagy ez ennek megfelelő konstans. Például:

```
type
  TForm1 = class(TForm)
    ...
    procedure WMMinMax (var Message: TMessage);
      message WM_GETMINMAXINFO;
end;
```

Az eljárás neve és a paraméter típusa rajtunk múlik, de a különböző Windows üzenetekre létezik sok előre definiált rekordtípus. Ez a technika különösen hasznos lehet veterán Windows programozóknak, akik mindent tudnak a Windows üzenetekről és az API függvényekről. A 10. fejezetben lesz az a példa (MinMax néven), amiből a fenti részlet származik.

Megjegyzés

Az, hogy a C nyelvhez hasonlóan magunk is kezelhetjük a Windows üzeneteket, és meghívhatjuk az API függvényeket, egyeseket megijeszt, másokat örömmel tölt el, de a Delphiben csak ritkán van szükség üzenetek kezelésére. Általában összetett komponensek írásakor kell alacsony szintű üzenetekkel és API hívásokkal foglalkozni.

Absztrakt metódusok

Az absztrakt (`abstract`) kulcsszó olyan metódusokat deklarál, amiket csak az adott osztály alosztályaiban fogunk definiálni. Az absztrakt metódusok nem előzetes deklarációk, hanem teljes metódusok. Ha megpróbálunk egy ilyen metódusnak definíciót adni, a fordító hibát fog jelezni:

```
type
  AbstractClass = class
    function F: Integer; virtual; abstract;
end;
```

Az Object Pascal képes olyan osztályokból példányt létrehozni, amiknek absztrakt metódusai vannak, bár a Delphi 32 bites fordítója egy figyelmeztető üzenetet küld ilyen esetekben.

Megjegyzés

A C++-ban még szigorúbb megkötések vannak, ott absztrakt osztályoknak – amiknek csak virtuális metódusaik vannak – nem lehetnek egyedei.

Ha egy absztrakt metódust meghívunk, akkor a Delphi futási hibát generál és leállítja az alkalmazást. A legtöbb futási hibától eltérően a 201-es hibánál – „Absztrakt metódus hívása" – nem keletkezik kivétel. Egy absztrakt metódus hívása (olyan metódusé, aminek nincsen kifejtése) olyan súlyos hibának számít, amire mindenképpen ki kell lépni a megfelelő kivétel után.

Eszünkbe juthatott, hogy miért is kell nekünk absztrakt metódus. Az ok a polimorfizmusban rejlik. Ha a TAnimal osztályban a Verse absztrakt metódus, akkor azt minden alosztály újradefiniálhatja. Az előny, hogy az általános MyAnimal objektum segítségével minden alosztály segítségével definiált állatra hivatkozhatunk, és meghívhatjuk ezt a metódust. Ha ez a metódus nem szerepelt volna a TAnimal osztály interfészében, akkor a fordító, ami statikus típusellenőrzést végez, nem engedte volna meg a hívást. Más szóval, az általános MyAnimal objektum hívásakor csak olyan metódusokat hívhatunk, ami a saját osztálya, a TAnimal definiál.

Tehát nem hívatunk alosztályok által létrehozott metódusokat, ha azoknak legalább a deklarációja nem szerepelt a szülő osztályban, például absztrakt metódus formájában. A következő példa az Animals3, amely bemutatja az absztrakt metódusok használatát és a polimorfizmus néhány másik jellemzőjét. Az ANIM.PAS állomány új verziójában három osztályt deklaráltunk, a TAnimal, a TDog, és a TCat. Itt következik az osztályok interfésze:

```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
    function Verse: string; virtual; abstract;
  private
    Kind: string;
  end;
```

```
TDog = class (TAnimal)
public
  constructor Create;
  function Verse: string; override;
  function Eat: string; virtual;
end;

TCat = class (TAnimal)
public
  constructor Create;
  function Verse: string; override;
  function Eat: string; virtual;
end;
```

A definíció legérdekesebb része a TAnimal osztály, amiben van egy absztrakt virtuális metódus, a Verse. Azt is érdemes megnézni, hogy mind a két osztály felülírja ezt a definíciót, és ad egy új virtuális metódust, az Eat nevűt. Mi a különbség a két megoldás között? A Verse függvény meghívásához használhatjuk az előző verzióban alkalmazott kódrészletet:

```
LabelVerse.Caption := MyAnimal.Verse;
```

Hogyan hívhatjuk meg az Eat metódust? A TAnimal osztályra nem alkalmazhatjuk, hiszen a

```
LabelVerse.Caption := MyAnimal.Eat;
```

kifejezés eredménye a „Field identifier expected" (mezőazonosító szükséges) hibaüzenet lesz.

Ennek a leküzdésére használhatjuk a Futás Idejű Típus Információt (RTTI), hogy a TAnimal objektumot TCat vagy TDog objektummá változtassuk, de a megfelelő típuskonverzió nélkül a program kivételt eredményez. A következő részben lesz erre egy példa. A metódus definíció hozzáadása a TAnimal osztályhoz tipikus megoldása a problémának, és az absztrakt kulcsszó jelenléte még kedvez is ennek a döntésnek.

Hogy kipróbáljuk a három új osztályunkat, módosítsuk az előző változat formját, úgy, hogy az új állatfajtának egy új rádiógombot teszünk fel. A TCat osztályhoz szükséges kódon kívül még más változások is vannak.

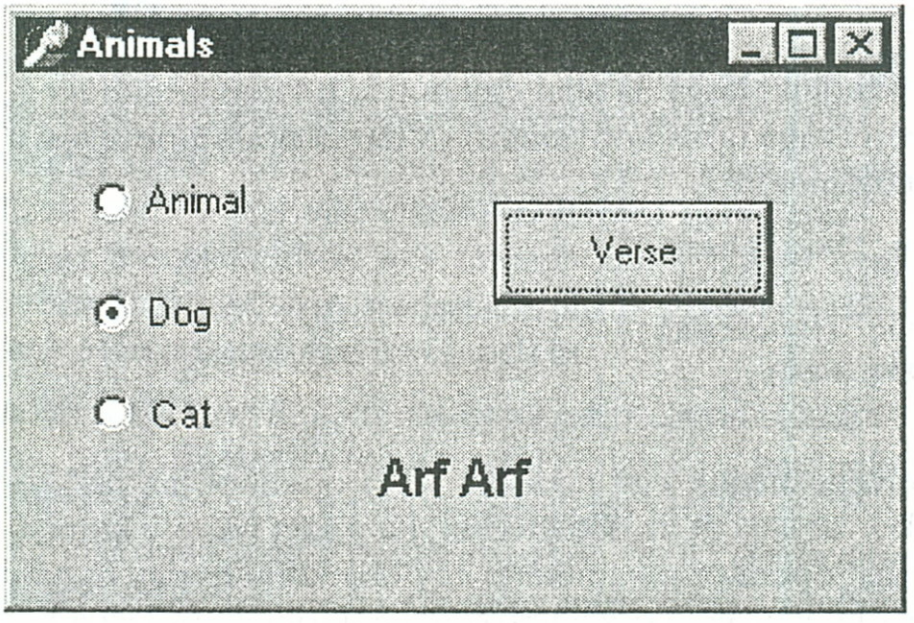
A program előző két változatában minden állattípusból készítettünk egy példányt, és a már létező objektumokat adtuk értékül a MyAnimal változónak, amikor a felhasználó kiválasztotta az egyik rádiógombot. Most egyszerre csak egy objektum létezik; a formnak csak egy private mezője van, a MyAnimal, amit az OnCreate eseménykezelőben a `MyAnimal := TDog.Create;` kifejezéssel inicializáltunk. Minden rádiógomb OnClick eseménye eltávolítja az aktuális objektumot, és újat készít:

```
procedure TFormAnimals.RbtnDogClick(Sender: TObject)
begin
  MyAnimal.Free;
  MyAnimal := TDog.Create;
end;
```

A VerseButtonClick metódus nem változik, még mindig a MyAnimal-t hívja, és a hang lejátszása mellett, a címkén megjeleníti a végeredményt. Amikor a programot futtatjuk, akkor a címke egy kutya, vagy egy macska hangját fogja mutatni (5.12. ábra), vagy súlyos futási hiba keletkezik, ami leállítja az alkalmazást, és még a rendszert is instabillá teheti. Ez akkor következik be, ha az Animal gombra kattintunk, és igy meghívjuk a TAnimal osztály absztrakt metódusát.

Új!

Az absztrakt osztályokon – amiknek vannak absztrakt metódusai – kívül a Delphi 3 lehetővé teszi olyan interfészek – pszeudo-osztályok definiálását is, amiknek csak absztrakt metódusai vannak. Az ilyen osztályokat szokták még absztrakt osztályoknak nevezni. Az interfészekről először a 6. fejezetben fogunk bővebben beszélni, de sokkal részletesebben lesz szó róluk a 21. fejezetben.



5.12. ábra *Az Animals3 példa képernyője a Cat rádiógomb kiválasztása után*

Futási idejű típus információ

Az Object Pascal leszármazott objektumokra vonatkozó típuskompatibilitási szabálya megengedi, hogy az ős osztály helyett a leszármazott osztályt használjuk, de ahogy azt korábban említettük, ennek az ellenkezője nem lehetséges.

Most tegyük fel, hogy a TDog osztálynak van egy Eat metódusa, ami nem szerepel a TAnimal osztályban. Ha a MyAnimal változó egy kutyára vonatkozik, akkor ezt a függvényt meghívhatjuk, ám ha megpróbáljuk, és a változó egy másik objektumra mutat, akkor az hibát okoz. A direkt típus átnevezéssel súlyos futási hibát okozhatunk – vagy, ami még rosszabb, értékes memóriaterületeket írhatunk felül, hiszen a fordító nem tudja eldönteni, hogy az objektum típusa valóban helyes-e, vagy, hogy a meghívott metódusok léteznek-e.

A probléma megoldására használhatunk futás idejű típus információn (RTTI) alapuló technikákat. Alapvetően, mivel minden objektum „tudja" a saját és a szülő osztályának típusát, ezért ezt az információt lekérdezhetjük az „is" operátor használatával, vagy a TObject osztály néhány metódusával (erről a következő fejezetben lesz szó). Az operátor paraméterei egy objektum, és egy osztály típus, a visszaadott érték pedig logikai:

```
if MyAnimal is TDog then
  ...
```

Ez a kifejezés csak akkor ad vissza igaz értéket, ha a MyAnimal objektum egy TDog, vagy a TDog osztályból származó osztályú objektumra mutat. Vagyis, ha azt vizsgáljuk, hogy a TDog objektum TAnimal típusú-e, akkor ez a teszt sikeres lesz. Más szavakkal, ez a kifejezés akkor igaz, ha az objektumot (MyAnimal) biztonságosan értékül adhatjuk egy, az adattípussal (TDog) megegyező típusú változónak.

Most, hogy már biztosan tudjuk, hogy az állat egy kutya, biztonságosan megtehetjük a típus átnevezést (vagy típus konverziót). A direkt típus átnevezést a következő módon tehetjük meg:

```
if MyAnimal is TDog then
begin
  MyDog := TDog (MyAnimal);
  Text := MyDog.Eat;
end;
```

Ugyanezt a műveletet elvégezhetjük egy második RTTI operátor, az „as" segítségével, ami csak akkor alakítja át az objektumot, ha a kért osztály kompatíbilis a jelenlegivel. Az „as" operátor paraméterei egy objektum és egy osztály típus, a végeredmény pedig egy, az új osztályra konvertált objektum. Ez a következőképp néz ki:

```
MyDog := MyAnimal as TDog;
Text := MyDog.Eat;
```

Ha csak az Eat funkciót akarjuk meghívni, akkor alkalmazhatjuk az alábbi, rövidebb jelölést:

```
(MyAnimal as TDog).Eat;
```

Ennek a kifejezésnek az eredménye egy, a TDog osztályba tartozó objektum, amire az osztály minden metódusát alkalmazhatjuk. A különbség a hagyományos, és az „as" konverzió között, hogy a második egy kivételt ad, ha az objektum típusa nem kompatíbilis azzal, amivé megpróbáltuk átalakítani. Ez az EInvalidCast kivétel (a kivételekről a következő fejezetben lesz szó).

Hogy elkerüljük ezt a kivételt, használjuk az „is" operátort, és ha sikerül, akkor használjuk az egyszerű átnevezést, hiszen semmi nem indokolja az „as" és az „is" egymás utáni használatát, ami dupla típusellenőrzést eredményezne:

```
if MyAnimal is TDog then
  TDog(MyAnimal).Eat;
```

Mind a két RTTI operátor nagyon hasznos a Delphiben, mert gyakran felmerül az az igény, hogy olyan általános kódot írjunk, amit több azonos, vagy eltérő típusú komponensben is felhasználhatunk. Amikor egy komponenst paraméterként átadunk egy eseménykezelő metódusnak, akkor az általános TObject típust használjuk, ilyenkor gyakran van szükség arra, hogy ezt eredeti típusára visszaalakítsuk:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Sender is TButton then
    ...
end;
```

Ez bevett gyakorlat a Delphiben, a könyv számos példájában találkozhatunk ilyennel. Az `is` és `as` operátorokon kívül az RTTI kifejezés még sok olyan műveletet takar, amit bármilyen osztályon vagy objektumon végrehajthatunk. Ezek közül a technikák közül néhányat bemutatunk a következő fejezetben, de az összetettebbek már túlmutatnak e könyv keretein, erről bővebben a Sybex által 1997-ben kiadott Delphi Developer's Handbookban olvashatunk, amit Marco Cantù és Tim Gooch írt.

A két RTTI operátor, az `is` és az `as`, nagyon sokoldalú, ezért úgy gondolhatjuk, hogy általános programozási eszközként használjuk őket, de ez nem javasolt. Több osztályt érintő összetettebb problémák megoldásánál először mindig próbáljuk meg a polimorfizmust. Az RTTI operátorokat akkor használjuk, ha a polimorfizmussal nem lehet megoldani a problémát, és ne a polimorfizmus helyett. Ez egyrészt rossz programozói szokás, másrészt lassabb programokat eredményez, mert az RTTI-nek erős negatív hatása van a program teljesítményére, hiszen végig kell járni az osztályhierarchiát, hogy megnézzük, hogy az értékadás helyes-e. Amint azt láthattuk, a virtuális metódusokhoz csak memóriacímzés kell, ami sokkal gyorsabb.

Kivételkezelés

Az Object Pascal utolsó érdekessége, amivel ebben a fejezetben foglalkozunk, a kivételkezelés. A kivételek mögött az az ötlet húzódik meg, hogy a programokat biztonságosabbá tehetjük, ha megadjuk a lehetőséget, hogy lekezeljék a hardver és szoftver hibákat. Az ilyen hibákat a program túlélheti, vagy rendesen kiléphet, megengedve a felhasználónak, hogy elmentse az adatait.

A hibák lekezelésére több módszer is adódik, ezek egyike a függvények visszatérési értékeinek vizsgálata. Az azonban, hogy mindig minden függvénynél megnézzük, hogy történt-e valami nemkívánt esemény, nem csak unalmas, de sok hibához is vezethet. A függvény visszatérési hibaértékek problematikusak, hiszen minden hívást le kell ellenőriznünk, hogy történt-e hiba, és ha igen, akkor ezt a rutint be kell fejezni, a foglalt erőforrásokat felszabadítani, és a hibát jelenteni kell a hívónak. A hívási lánc minden eleme felelős azért, hogy a hibainformációt továbbadja a fölötte lévőnek. Ha akár egyetlen eljárás is elmulasztja továbbadni a hibajelzést, akkor a kódunk nem fog tudni a hibáról, ami tönkreteheti az egész műveletet.

A kivételek ezt a problémát úgy oldják meg, hogy a továbbító mechanizmust kiveszik a normális kódból. Ha egy kódrészletnél nem számít az esetleges hiba, akkor annak nem is kell tennie semmit, hogy a hibajelentés jól működjön. A kivételek lehetővé teszik, hogy kompaktabb kódot írjunk, amit kevésbé darabolnak fel a fő céltól független karbantartási részek. A kivételek azt is megengedik, hogy a hibafelderítésre más kódot használjunk, mint ami a hibát jelzi.

A kivételek többletet jelentenek abból a szempontból is, hogy általános és egységes hibajelentési mechanizmust alkotnak, amit a Delphi könyvtárak – és maga a keretrendszer is – használ. Futásidőben, ha valami rossz történik, a Delphi egy kivételt vált ki. Ha a programunk jól van megírva, akkor tudomásul veheti a problémát, és megpróbálhatja megoldani; máskülönben a kivétel továbbkerül a hívóhoz, és így tovább. Ha egy kivételt nem kezel le a programunk, akkor a Delphi megteszi helyettünk azzal, hogy megjelenít egy szabványos hibaüzenetet, és megpróbálja folytatni a programot.

Az egész mechanizmus négy kulcsszón alapul:

- `try` vezeti be a védett kódblokkot.
- `except` zárja a védett blokkot, és vezeti a kivételkezelő kifejezéseket, amik a következőként néznek ki: `on` *`kivételtípus`* `do` *`kifejezés`*.
- `finally` jelzi azokat a programrészeket, amiket mindenképpen le kell futtatni, még kivételek esetén is.
- `raise` segítségével generálhatunk kivételeket. A legtöbb kivételt, amivel a Delphi programozás során találkozunk, a rendszer generálja, de

akár saját kivételeket is emelhetünk, ha a programunk futásidőben vesz észre hibás, vagy inkonzisztens adatokat. A `raise` kulcsszót használhatjuk még arra, hogy egy kezelőben újra létrehozzuk a hibát, ezzel továbbadva azt a következő kezelőnek.

Íme egy példa egy egyszerű védett blokkra:

```
function Divide (A, B: Integer): Integer;
begin
  try
    {the following statement is protected because
    it can generate an error if B equals 0}
    Result := A div B;
  except
    on EDivByZero do
      Result := 0;
  end;
end;
```

A kivételkezelő részben az EDivByZero (nullával való osztás) kivételt fogjuk el, amit a Delphi definiál. Több ilyen kivétel van, ami futásidejű problémákra vonatkozik, mint például a nullával való osztás, vagy a rossz dinamikus típusváltoztatás, és vannak komponens hibákra vonatkozóak, mint például a rossz index. A programozók saját hibákat is definiálhatnak. Csak annyit kell tennünk, hogy az alapértelmezett kivétel osztályból, vagy valamelyik alosztályából új osztályt származtatunk:

```
type
  EArrayFull = class (Exception);
```

Amikor új elemet adunk egy tömbhöz, és az már tele van – valószínűleg egy program logikai hiba miatt – akkor a megfelelő kivételt emelhetjük úgy, hogy elkészítjük ennek az osztálynak egy objektumát.

```
if MyArray.Full then
  raise EArrayFull.Create ('Array full');
```

Ez a Create metódus, amit az Exception osztályból örököltettünk, egy stringet kap paraméterként, ami a felhasználó felé jelzi a hibát. Az így elkészített kivétel objektum megsemmisítésével nem kell törődnünk, mert ezt automatikusan megteszi a kivételkezelő mechanizmus.

Az at kulcsszó

Van még egy, az eseménykezeléssel összefüggő, bár nagyon ritkán használt kulcsszó, az `at`. Ezt a kulcsszót a `raise` kifejezésben használhatjuk, hogy jelezzük, hogy melyik gépi kódú rész jelenjen meg a kivétel keletkezési helyeként.

A szintaxis a következő:

```
raise object at location;
```

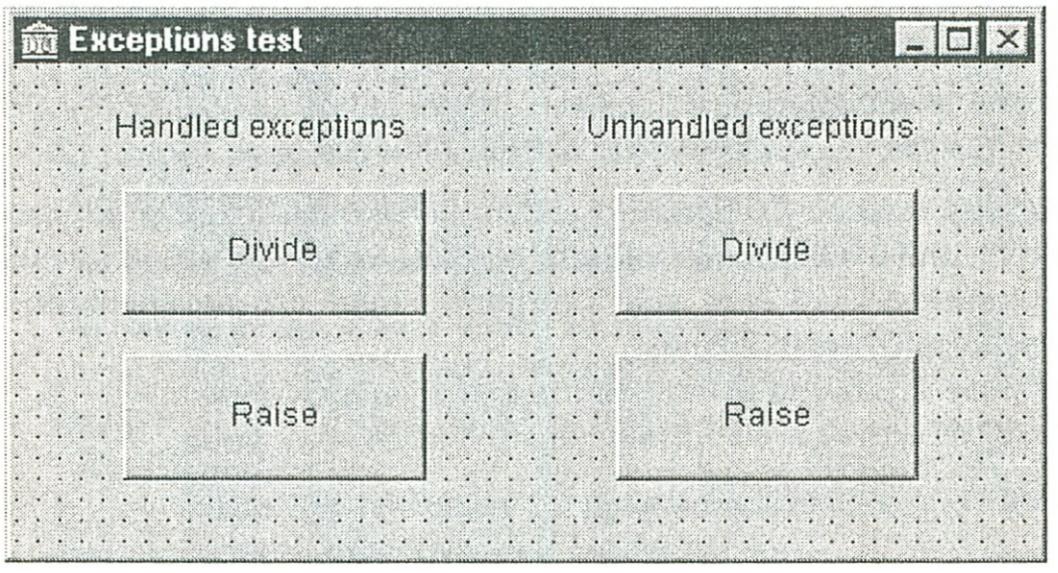
Például a SYSUTILS.PAS forrásában van a következő kifejezés:

```
raise OutOfMemory at ReturnAddr;
```

Ez a hiba úgy jelentkezik, mintha a hívó programban keletkezett volna, és nem az eljárásban, ezt jelenti a ReturnAddr rendszer függvény által visszaadott hely. Ez értesíti a futtató modult, hogy a kivételt a hívóban kell generálni. Ilyen kódnak természetesen csak nagyon speciális esetekben van értelme.

Példa kivételek használatára

Az ebben a példában bemutatásra kerülő kód, valamint az előző Divide függvény egy olyan program részei, ami arra készült, hogy teszt kivételekkel kísérletezhessünk. A programot Except-nek hívják, és az 5.13. ábrán látható formot jeleníti meg. A formon négy gomb van, mindegyik jellemző névvel, és egy OnClick eseménykezelővel.



5.13. ábra *Az Except példa formja tervezés közben*

Akárhányszor a felhasználó megnyomja az egyik gombot, egy kivétel képződik, vagy nullával való osztás, vagy egy direkt raise kifejezés miatt. A bal oldali két gombnak helyes kivétel kezelési kódja van, míg a jobb oldaliaknál nem foglalkoztunk a kivételek kezelésével. A különböző gombok lenyomásakor megvizsgálhatjuk, hogyan viselkedik a program.

Tipp

Amikor egy programot a hibakeresőben futtatunk, az megállítja a programot kivétel esetén. Általában ez az, amit akarunk, hiszen innen tudhatjuk, hogy hol keletkezett a kivétel, és lépésről-lépésre megnézhetjük a kezelő hívását. Ebben a tesztprogramban azonban ez a viselkedés összezavarja a program végrehajtását. A valóságban a debugger még a jól lekezelt kivételeknél is leáll, innen lépésenként haladva megnézhetjük, hogy azokat hogyan kezeli le a kód. Ha azt szeretnénk, hogy jól lekezelt kivételek esetében a program ne álljon le, akkor futtassuk a Windows Intézőből, vagy átmenetileg kapcsoljuk ki a Delphi **Environment Options** (környezeti beállítások) menüpontjában a **Break on Exception** (megállás kivételnél) szolgáltatást.

Az első Divide (osztás) gomb a Divide függvény egy, az előbb leírttól kissé eltérő változatát hívja meg:

```
procedure TForm1.ButtonDivide1Click(Sender: TObject);
begin
  Divide (10, 0);
end;

{protected version of the div operator}
function Divide (A, B: Integer): Integer;
begin
  try
    {the following statement is protected because it
    can generate an error if B equals 0}
    Result := A div B;
  except
    on EDivByZero do
    begin
      Result := 0;
      MessageDlg ('Divide by zero corrected',
        mtError, [mbOK], 0);
    end;
```

```
      on E: Exception do
      begin
        Result := 0;
        MessageDlg (E.Message,
          mtError, [mbOK], 0);
      end;
    end; // end except
  end;
```

Ez a program egy kivételt eredményez, amit azonnal el is fog. Figyeljük meg, hogy ugyanabban a try blokkban két különböző kivételkezelő is helyet kapott. Egy blokkban akárhány kivételkezelőt elhelyezhetünk, ezek fentről lefele kerülnek kiértékelésre, ezért az átfogóbb kezelőket – a kivéve ős osztályok kezelőit – a végére kell tennünk.

Ténylegesen, a kivételek hierarchiájának figyelembe vételével a kezelő meghívódik minden olyan esetben is, amikor az általa hivatkozott kivétel egy alosztályáról van szó, akár bármelyik eljárás. Ismét akcióban a polimorfizmus. Azt azonban jegyezzük meg, hogy minden kivételt lekezelni, ahogy az a példában látszik, általában nem jó döntés, jobb a Delphire hagyni az ismeretlen kivételeket. A VCL alapértelmezett kivételkezelője megjelenít egy üzenetablakot a kivétel hibaüzenetével, és utána folytatja a programot.

Másik fontos dolog, hogy a kezelő által kapott kivétel objektumot felhasználhatjuk. A fenti példában van egy ilyen részlet:

```
on E: Exception do
  MessageDlg (E.Message, mtError, [mbOK], 0);
```

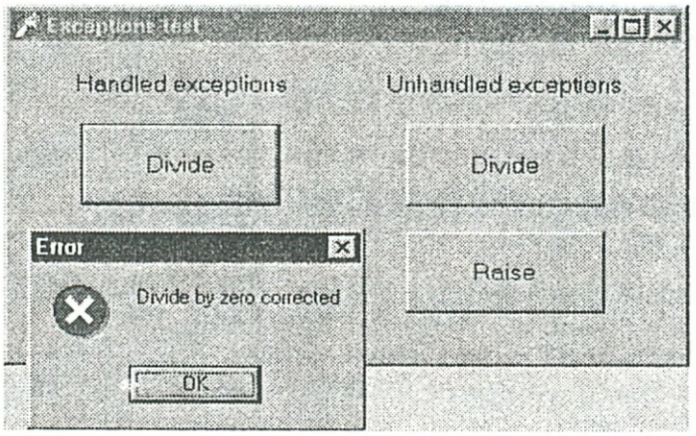
Az Exception osztályba tartozó E objektum felveszi a raise által elküldött kivétel objektum értékét. Amikor kivételekkel dolgozunk, mindig tartsuk szem előtt, hogy kivétel generálásakor tulajdonképpen objektumot készítünk, és hogy a típusának megnevezésével kezeljük. Ez fontos előny, hiszen, ahogy azt láthattuk, amikor egy kivétel típust lekezelünk, akkor a valóságban azt a típust, és az összes leszármazottját is kezeljük.

Tipp

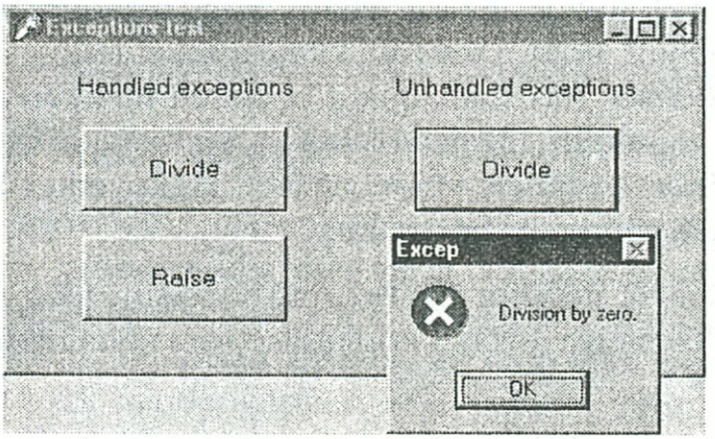
A Delphi hierarchikus kivételeket definiál, ezért kezelhetjük mindegyik kivétel típust külön, vagy akár csoportonként együtt is. A 7. fejezetben vagy egy ábra, ami a Delphi kivételek hierarchiáját ábrázolja.

Az 5.14. ábrán látható üzenet ablak csak azért jelenik meg, mert a MessageDlg hívással elkészítettük. Egyébként, ha csak simán nullával osztunk, akkor a Delphi az 5.15. ábrán látható szabványos üzenetablakot jeleníti meg. Ez történik a második Divide gomb megnyomásakor:

```
procedure TForm1.ButtonDivide2Click(Sender: TObject);
var
  A, B, C: Integer;
begin
  A := 10;
  B := 0;
  {generates an exception, which is not handled}
  C := A div B;
  {we have to use the result, or the optimizer will
  remove the code and the error, too}
  Caption := IntToStr (C);
end;
```



5.14.ábra *Az Except példa első Divide gombja kivételt generál, és le is kezeli azt*



5.15. ábra *Az Except példa második Divide gombja kivételt generál, de a kezelést a Delphire hagyja*

A másik két gomb még egyszerűbb, csak meghívnak egy olyan függvényt, ami mindenképpen kivételt okoz. Az első gomb lekezeli, anélkül, hogy a felhasználónak akár egy üzenetablakot is mutatna, a második pedig nem figyel a kivételekre. Amikor a felhasználó megnyomja az első Raise gombot látszólag semmi sem történik. Ez az, amit úgy hívunk, kivételkezelés!

```
{fake procedure: the array is always full}
procedure AddToArray (N: Integer);
begin
  raise EArrayFull.Create ('Array full');
end;

procedure TForm1.ButtonRaise1Click(Sender: TObject);
begin
  try
    // this procedure raises an exception
    AddToArray (24);
  except
    // simply ignores the exception
    on EArrayFull do; // do nothing
  end;
end;

procedure TForm1.ButtonRaise2Click(Sender: TObject);
begin
  // unguarded call
  AddToArray (24);
end;
```

Kivételek és a verem

Amikor a programban kivétel keletkezik, és az aktuális rutin nem kezeli le, akkor mi történik a hívási veremmel (a call stackkal)? A program elkezd keresni egy kezelőt a vermen lévő függvények között. Ez azt jelenti, hogy a program kilép a meglévő függvényekből, anélkül, hogy a további kifejezéseket lefuttatná. Hogy megértsük ennek a működését, használhatjuk a beépített hibakeresőt, vagy elhelyezhetünk néhány egyszerű üzenetablakot a kódban, hogy tudjuk, bizonyos forráselemek mikor kerülnek végrehajtásra. A következő példában, az Except2-ben, a második módszert használtuk.

Például, amikor megnyomjuk a Raise2 gombot az Except2 formján, akkor olyan kivétel keletkezik, ami nincs lekezelve, ezért a kód utolsó sora soha nem fut le:

```
procedure TForm1.ButtonRaise2Click(Sender: TObject);
begin
  // unguarded call
```

```
    AddToArray (24);
    ShowMessage ('Program never gets here');
end;
```

Ez az AddToArray eljárást hívja, amelyik a kivételt okozza, ennek ez eljárásnak az utolsó sora sem fut le:

```
procedure AddToArray (N: Integer);
begin
  raise EArrayFull.Create ('Array full');
  ShowMessage ('Program never gets here');
end;
```

Ha az Except2 programot futtatjuk, akkor soha nem láthatjuk az üzeneteket, hacsak nem tesszük megjegyzésjelek közé a raise kifejezést, ami azért javasolt, hogy lássuk a különbséget aközött, amikor van kivétel, és amikor nincs.

Amikor egy kivételt lekezelünk, akkor a program futása a kezelő után folytatódik, és nem a kivételt okozó sornál. Ezt mutatja ez a módosított metódus:

```
procedure TForm1.ButtonRaise1Click(Sender: TObject);
begin
  try
    // this procedure raises an exception
    AddToArray (24);
    ShowMessage ('Program never gets here');
  except
on EArrayFull do
      ShowMessage ('Handle the exception');
  end;
  ShowMessage ('Exception has already been handled');
end;
```

Az utolsó ShowMessage hívás a második után jelenik meg, az első azonban sohasem. A programot futtassuk, módosítsuk a kódját, és kísérletezzünk vele, ameddig meg nem értjük a program futásának menetét kivétel esetén. A teljes példát a kísérő lemezen Except2 néven találhatjuk meg.

A finally blokk

A kivételkezelésnél van még egy negyedik kulcsszó, amit ugyan említettünk, de amiről eddig még nem volt szó, és ez a `finally`. A `finally` blokkot olyan tevékenységek elvégzésére használjuk, amiket mindenképpen meg kell tenni, tipikusan ilyenek az erőforrás felszabadítási műveletek. A `finally` blokk mindenképpen végrehajtásra kerül, attól függetlenül, hogy volt-e kivétel, vagy nem, eltérően a `try` blokkot követő sima kódtól, ami csak akkor fut le, ha nem volt kivétel. Ezt szemlélteti a következő függvény:

```
function ComputeBits (A, B: Integer): Integer;
var
  Bmp: TBitmap;
begin
  try
    Bmp := TBitmap.Create;
    // compute bits ...
    Result := A div B;
    Bmp.Free;
  except
    on EDivByZero do
    begin
      Result := 0;
      MessageDlg ('Error in ComputeBits',
        mtError, [mbOK], 0);
    end;
  end; // end except
end;
```

Ez a kód alapvetően hibás, hiszen, ha B nulla, akkor kivétel keletkezik, és a képet felszabadító utasítás soha nem kerül végrehajtásra. A program a kivételt generáló sorról a megfelelő kezelőre ugrik, kihagyva a köztes sorokat.

Arra gondolhatnánk, hogy ha az erőforrást felszabadító sort a függvény végére tesszük, a `try` blokk után, megoldódik a probléma. De ha egy lekezeletlen kivétel keletkezik, akkor a `try` blokk sem fut le. Erre megoldás a `finally` kifejezés:

```
function ComputeBits (A, B: Integer): Integer;
var
  Bmp: TBitmap;
begin
  Bmp := TBitmap.Create;
  try
    // compute bits ...
    Result := A div B;
  finally
    Bmp.Free;
  end; // end finally
end;
```

Amikor a program lefuttatja a függvényt, az kivételektől függetlenül mindenképpen meghívja a TBitmap (kép) objektum Free metódusát. Ennek a verziónak megvan az a hátránya, hogy a kivételeket nem kezeli le. Furcsa módon ez nem lehetséges, mert a `try` blokkot követheti egy `except`, vagy egy `finally` kifejezés, de mindkettő nem állhat egyidőben. Erre megoldás két egymásba ágyazott `try` blokk használata, ahol a belsőben van a `finally`, és

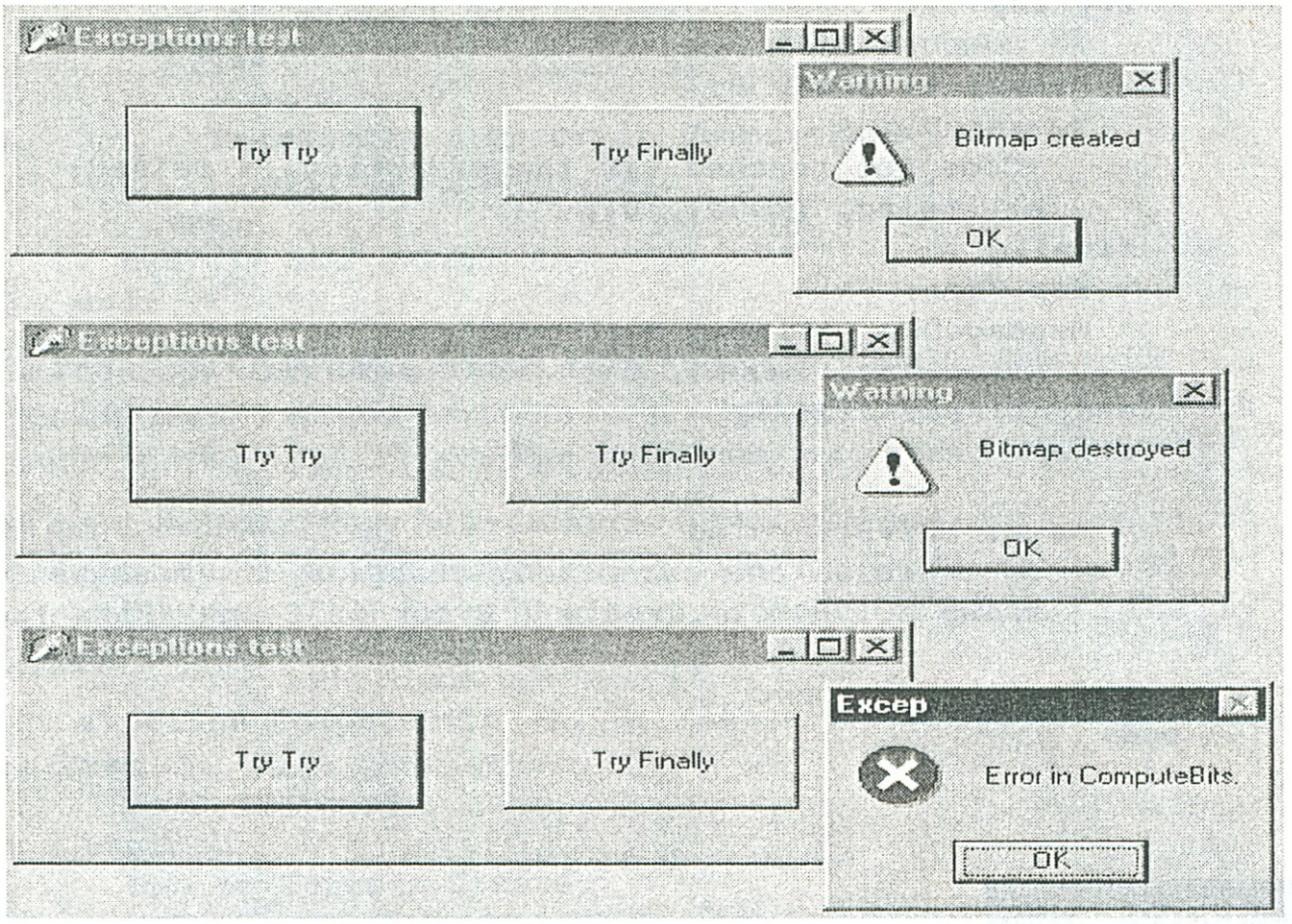
a külsőben az **except**, vagy fordítva, ahogy azt az adott helyzet megköveteli. Erre példa a következő:

```
function ComputeBits (A, B: Integer): Integer;
var
  Bmp: TBitmap;
begin
  Bmp := TBitmap.Create;
  MessageDlg ('Bitmap created',
    mtWarning, [mbOK], 0);
  try try
    // compute bits ...
    Result := A div B;
    MessageDlg (
      'Code not reached (if the exception is raised)',
      mtWarning, [mbOK], 0);
  finally
    Bmp.Free;
    MessageDlg ('Bitmap destroyed',
      mtWarning, [mbOK], 0);
  end; // end finally
  except

    on E: EDivByZero do
      // re-raise the exception, changing its message
      raise EDivByZero.Create ('Error in ComputeBits');

  end; // end except
end;
```

Figyelem!

Elméletben lehetséges lenne, hogy megváltozassuk a kivétel objektum Message (üzenet) paraméterét pl. egy E.Message := 'Hiba ComputeBits-ben!' kifejezéssel, és azután a **raise** segítségével továbbadjuk ezt a kivétel objektumot további paraméterek nélkül. Azonban a Delphi 32 bites változatában, a környezettől és a verziótól függően, ez nem mindig működik. Ezért a „hivatalos" megoldás egy ugyanolyan típusú új objektum létrehozása, mint a fenti kódban. Ez így legalább mindig működik.

Hogy megmutassuk, ez a kód úgy működik, ahogy azt fentebb leírtuk, egy programot építettünk a függvény köré. Ez az új példa az Except3 nevet kapta, és csak két gombja van – az 5.16. ábra azt mutatja, amikor a felhasználó az első gombra kattint. A kód a semmit sem csináló ComputeBits függvény két változatát tartalmazza. Az egyik megegyezik a fentivel, a másikban csak `finally` blokk van. Mivel a függvény több üzenet ablakot jelenít meg, így tudhatjuk, hogy a kép mikor készül el, mikor semmisül meg, és hogy hogyan fut a program.



5.16. ábra *Az Except3 program három üzenetablaka, miután az első gombot megnyomtuk. Figyeljük meg a futás sorrendjét*

A blokkokat célszerű `finally` kifejezésekkel védeni, hogy elkerüljük az erőforrás vagy memória lyukakat kivételek esetén. A kivételek lekezelése talán kevésbé fontos, hiszen a Delphi túléli a legtöbbet. Az Except3 két ComputeBits függvénye között annyi a különbség, hogy az első saját hibaüzenetet ad, az előbb látott kóddal. Hogy megértsük a programot, futtassuk, nézzük meg, mit csinál, és próbáljuk megváltoztatni a viselkedését. Nézzük meg a 5.16. ábrát, ahol három üzenet látható, miután a felhasználó megnyomta a Try gombot.

Hogyan tovább?

Ebben a fejezetben megismertük az objektum orientált programozást Object Pascalban. Meghatároztuk az osztály definícióját, megnéztük a metódusok használatát, az információ elrejtést, az öröklést, az egységek definícióját, a futás idejű típus információt, és a kivételkezelést. Ez valóban elég sok újdonság egyetlen fejezetben, különösen, ha most találkoztunk először az OOP-vel, akkor ez lehet, hogy nehezen emészthető volt, de az OOP rövid áttekintését megtalálhatjuk az A függelékben, és javasolt egy olyan, a témával foglalkozó könyv áttanulmányozása is, amiben az elmélet is szerepel. Ahhoz, hogy Delphiben programozhassunk ez a fejezet elég lesz. Valószínűleg még érdemes lesz visszatérni ehhez és a következő két fejezethez, miután elolvastuk a második részt, ami a „Komponensek használata" címet viseli.

Ha igazán nagy gyakorlatot akarunk szerezni a Delphiben, akkor a következő két fejezetet figyelmesen olvassuk el, mert ezek bevezetnek az Object Pascal sok összetettebb részébe, a VCL felépítésébe, és bemutatnak néhány új Delphi 3 jellemzőt is.

6. FEJEZET

Haladó szintű Object Pascal

E fejezet tartalma:

- A Self kulcsszó
- Komponensek dinamikus létrehozása
- Osztálymetódusok és adatok
- Egy osztály objektum-számlálóval
- Metódus-mutatók
- A számlálós példa fejlesztése
- Osztályreferenciák
- Objektumok létrehozása futási idő alatt osztályhivatkozások használatával
- Interfész és többszörös öröklődés
- Tulajdonságok és események definiálása
- Események a Delphiben

Haladó szintű Object Pascal

Az előző fejezetben az Object Pascal, mint objektumorientált nyelv alapvető funkcióiról volt szó. Megbeszéltük az osztályokat, az öröklődést és a polimorfizmust (többalakúság), és az OOP nyelvek létrejöttét. Ez a fejezet az Object Pascal összetettebb témáival foglalkozik, és megmutatja, hogy lett a nyelv kibővítve a komponensfejlesztéssel és a COM modellel (az OLE alapja). A szolgáltatások közül, amelyekről ebben a fejezetben lesz szó, néhány teljesen új, és csak a Delphi 3-ban található meg; több más, kevésbé ismert funkció már megtalálható volt a Delphi 2-ben is.

Ajánlatos ezt a fejezetet is végigolvasni, ámbár néhány témakör tartalmát valószínűleg nem tudjuk rögtön alkalmazni, de segítenek a Delphiben írt programok és az operációs rendszer együttműködését megérteni. Ez és a következő fejezet valójában a Delphi komponens könyvtárának az Object Pascal nyelvre való felépítésével, és annak kibővítésével foglalkozik.

Megjegyzés

Ha nem vagyunk jártasak Delphiben és az objektumorientált programozásban, akkor egyelőre kihagyhatjuk ezt (és valószínűleg a következő) fejezetet. A könyv harmadik részének elérése előtt visszatérhetünk ide, mely rész főleg a komponensek fejlesztésével, és az OLE/COM technológiával foglalkozik.

A Self kulcsszó

A Self kulcsszó egy implicit paraméterre vonatkozik, amely automatikusan átadódik bármelyik meghívott metódusnak. A Selfet úgy definiálhatjuk, mint egy mutatót az aktuális objektumra (az osztály aktuális egyedére), és ezt a mutatót használja a metódus az adott objektum mezőinek elérésére. Valójában, amikor több ugyanolyan osztályú objektumot is deklarálunk, minden alkalommal, amikor meghívunk egy metódust, akkor az csak az adott objektum adataival dolgozik, a többi objektumra nincs hatással.

Például itt egy metódus, amivel az előző fejezetben már találkoztunk:

```
procedure Date.SetValue(m, d, y: Integer);
begin
  Month := m;
  Day := d;
  Year := y;
end;
```

Egy ilyen metódusban a Month az aktuális objektum Month mezőjére vonatkozik, amit így is kifejezhettünk volna:

```
Self.Month := m;
```

Gyakorlatilag a Delphi így fordítja le a kódot, nem úgy, ahogy azt nekünk kell beírni. A Self kulcsszó egy alapvető nyelvi elem, amelyet a fordító használ, de néha a programozók is, hogy megoldják a nevekkel járó problémákat, és a kódot olvashatóbbá tegyék. Ha valaha is használtunk valamilyen objektumorientált nyelvet, akkor már bizonyára találkoztunk a Self fogalmával, talán egy más néven (például C++ -ban ezt This-nek nevezik).

Amit tudnunk kell a Selfről az annyi, hogy a metódusok egy rejtett paramétere az Object Pascal nyelvben, és ez különbözteti meg a metódusokat az eljárásoktól és a funkcióktól, amelyek nem kapcsolódnak osztályokhoz. A metódus meghívása technikailag különbözik egy általános szubrutin meghívásától. A metódusoknak van egy plusz rejtett paramétere, a Self. De mivel mindez a háttérben történik, ezért most még nem kell többet tudnunk a Self működéséről.

Tipp

Ha megnézzük a VCL TMethod adattípusának definícióját, akkor láthatjuk, hogy ez egy rekord, egy Code mezővel, és egy Data mezővel. Az első egy mutató, amely a funkció kezdőcímére mutat, a második a metódus Self paraméterének értéke. A metódus-mutatókat hamarosan megbeszéljük.

Komponensek dinamikus létrehozása

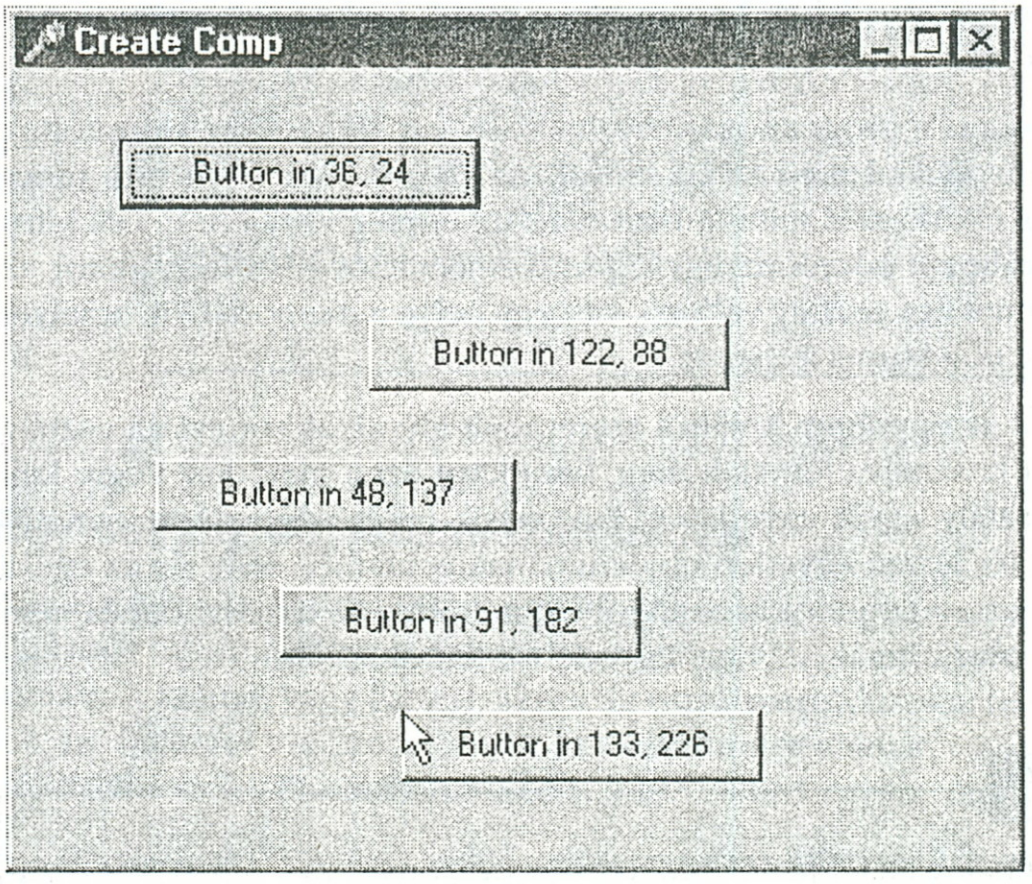
Delphiben a Self kulcsszót gyakran használjuk, amikor az aktuális form egyik metódusában kifejezetten a formra kell hivatkoznunk. Ennek tipikus példája, ha futási időben kell létrehoznunk egy komponenst, mert a Create konstruktorának át kell adni, hogy ki a „tulajdonosa", és azt az értéket kell a Parent (ős) tulajdonságához is hozzárendelni (a tulajdonos, és az ős közötti különbséget a következő fejezetben tárgyaljuk). Mindkét esetben az aktuális formot, mint paramétert és értéket kell átadnunk, és ennek a legjobb módja a Self kulcsszó használata.

Egy ilyen kód demonstrálásaképp írtam egy CreateC példát (CreateC – komponens létrehozására utal). Ez a program egy üres formból, és az OnMouseDown metódus kezeléséből áll. Azért használtam az OnMouseDownt az OnClick helyett, mert paraméterként szükség van az egérgomb lenyomásának pozíciójára.

Így oda rakunk le egy gomb komponenst, ahova kattintottunk. Itt a metódus kódja:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  Btn: TButton;
begin
  Btn := TButton.Create (Self);
  Btn.Parent := Self;
  Btn.Left := X;
  Btn.Top := Y;
  Btn.Width := Btn.Width + 50;
  Btn.Caption := Format ('Button in %d, %d', [X, Y]);
end;
```

Ez a kód az egér-kattintások helyén gombokat hoz létre, melyek címkéje a pontos helyzetüket tartalmazza, ahogy az a 6.1. ábrán is látszik. A fenti kódban jól láthatjuk, hogy a Self kulcsszót mint paramétert használtuk a Create metódusnál, és mint értéket a Parent tulajdonságnál.



6.1. ábra *A CreateC példa futásidő alatt létrehoz egy Button komponenst*

Figyeljük meg, hogy az előbbi metódushoz hasonló kódokban, gyakran használják a with kifejezést, ahogy azt a következő listában láthatjuk:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  with TButton.Create (Self) do
  begin
    Parent := Self;
    Left := X;
    Top := Y;
    Width := Width + 50;
    Caption := Format ('Button in %d, %d', [X, Y]);
  end;
end;
```

Osztálymetódusok és osztályadatok

Amikor egy osztályban definiálunk egy mezőt, akkor gyakorlatilag az a mező az osztály összes egyedéhez is hozzáadódik. Minden egyednek megvan a saját, független reprezentációja (amelyre a Self mutatóval hivatkozhatunk). Néhány esetben azonban hasznos lehet, ha van olyan mező, melyet az osztály összes objektuma megoszt.

Más objektumorientált programozási nyelvekben ezt formálisan valósították meg, a C++ -ban például a `static` szóval. Az Object Pascalban erre nincs szükség, mert az elkülönítés mindig biztosított az egység szintjén. Ebből kifolyólag, ha egy változót adunk az egység implementációs részéhez, akkor az úgy viselkedik, mint egy osztály változó, van egy helye a memóriában, amelyet az osztály összes objektuma megoszt.

Ha ezt a változót az egységen kívülről szeretnénk elérni, akkor ezt az osztály egyik metódusát használva tehetjük meg, bár ekkor ezt a metódust végre kell hajtatnunk az osztály egyik egyedével. Egy másik lehetőség az osztálymetódus, amely nem fér hozzá egyetlen objektumának adataihoz sem, de az egész osztályra vonatkozó adatokhoz igen. Az ilyen metódus az osztályhoz, és nem annak valamely objektumához, vagy egyedéhez kapcsolódik.

Tipp

Technikailag az Object Pascal osztálymetódusában a rejtett Self paraméter nem az osztály egyik objektumára hivatkozik (ahogy egy átlagos metódusban), hanem magára az osztályra. Ezt osztályhivatkozásnak nevezik.

Az osztálymetódusok több objektumorientált nyelvben megtalálhatóak, a C++ például static függvényekkel implementálja e funkcionalitást. Az osztálymetódust az Object Pascalban a function, vagy a procedure szavak elé írt class szóval deklarálhatjuk:

```
type
  MyClass = class
    class function ClassMeanValue: Integer;
```

Nem túlságosan elterjedt az osztálymetódusok használata az Object Pascalban, mert ugyanezt elérhetjük, ha egy osztályt deklaráló egységhez hozzáadunk egy függvényt, vagy eljárást, az igazi objektumorientáltság azonban ezt részesíti előnyben. Valójában a VCL meglehetősen gyakran használ osztálymetódusokat, ámbár van sok globális szubrutinja is. Az osztálymetódusok lehetnek virtuálisak is, amely lehetővé teszi fölülírásukat, így a polimorfizmust is.

Megjegyzés

A teljesen objektumorientált programozási nyelvekben, mint például a Javában vagy a SmallTalkban, a globális eljárások nem megengedettek, ehelyett mindig osztálymetódusokat kell használnunk. Habár ezt tartják jó objektumorientált megközelítésnek, ettől nem lesz igazán érthetőbb a kód. Ellenkezőleg, ez a programozókat arra kényszeríti, hogy keverjék az OOP kódot a nem OOP kóddal, és egyébként haszontalan osztályokat hozzanak létre, hogy legyen egy elérhető osztálymetódus. Ez inkább hátrány az ilyen nyelveknél, mint előny.

Egy osztály objektum-számlálóval

Amikor egy osztályadat (vagy egy egység adata, hogy pontosabbak legyünk) általános információkat tartalmaz az osztályról (mint például az osztály objektumainak száma, vagy listája), akkor az osztálymetódusokat használva elérhetjük azt (például visszanyerhetjük az objektumok számát, vagy azok listájában nézelődhetünk). Pontosan ez a célja a következő egyszerű példának is.

A CountObj program az előző példaprogram (CreateC) kiterjesztése. Az új form még most is meglehetősen csupasz, de új programsorokat tartalmaz. Gyakorlatilag egy teljesen új osztályt adtunk hozzá, amely a VCL TButton osztályából öröklődött, és egy új funkcióval bővült, az objektum-számlálással. Itt az új osztály deklarációja:

```
type
  TCountButton = class (TButton)
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    class function GetTotal: Integer;
  end;
```

Megjegyzés

Hogy pontosabbak legyünk, ebben a példában fölül kellene írnunk (override) a NewInstance metódust, a Create konstruktor fölülírása helyett. Valójában lehetséges egy már létrehozott objektum Create konstruktorának többszöri meghívása azért, hogy megismételjük az inicializálást egy új objektum létrehozása nélkül. Ezt egy már létrehozott objektum konstruktorának használatával érhetjük el (mint például a MyObject1.Create-ben), egy osztályé helyett (mint a MyObject1:=TMyClass.Create-ben).

Minden egyes alkalommal, amikor egy új objektumot hozunk létre, az alaposztály konstruktorának meghívása előtt a program megnövel egy számlálót. Minden alkalommal, mikor egy objektum megsemmisül, a számlálót csökkentjük:

```
constructor TCountButton.Create (AOwner: TComponent);
begin
  inherited;
  Inc (TotBtns);
end;

destructor TCountButton.Destroy;
begin
  Dec (TotBtns);
  inherited Destroy;
end;
```

Tipp

Elméletileg, amikor a Create konstruktor örökített verzióját hívjuk meg, akkor újra meg kell adnunk a metódus nevét és paramétereit. Gyakorlatilag a fenti rövid kód ugyanezt teszi! Közvetlenül használtuk a szokásos megközelítést a destruktorban, egyszerűen azért, mert nincsenek paraméterei, és egyszerűbb leírni.

A számláló egy változó, amelyet az egység implementációs részében deklaráltunk, és külső egységek számára elérhetetlen. Egyedül az osztálymetódussal olvashatjuk ki az aktuális értékét:

```
implementation

var
  TotBtns: Integer = 0;

class function TCountButton.GetTotal: Integer;
begin
  Result := TotBtns;
end;
```

Ezt a változót inicializálhatjuk már a definiálásakor is.

Most már ebből az új típusú objektumból létrehozhatunk újabbakat, a FormMouseDown metódust kissé megváltoztatva.

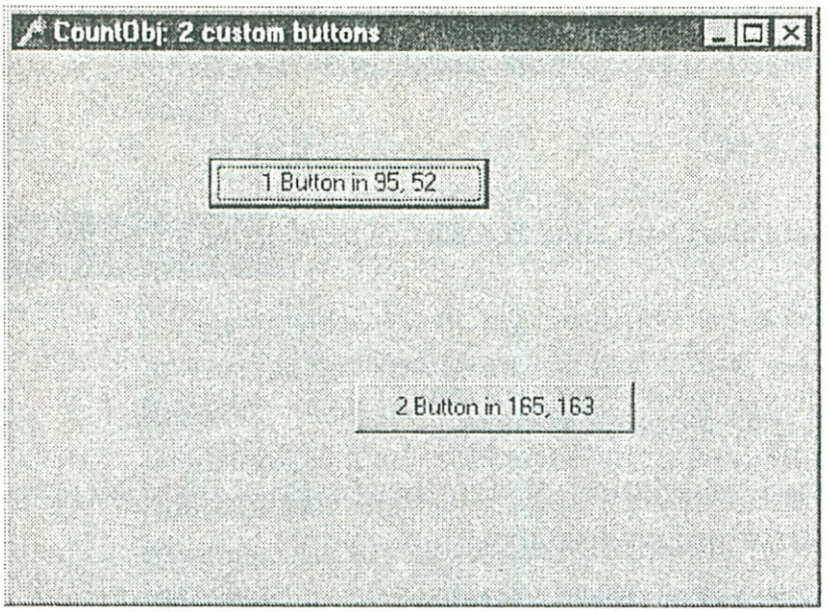
```
procedure TForm1.FormMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  with TCountButton.Create (Self) do
  begin
    Parent := Self;
    Left := X;
    Top := Y;
    Width := Width + 60;
    Caption := Format ('%d Button in %d, %d',
      [GetTotal, X, Y]);
  end;
end;
```

Minden alkalommal, ha egy TCountButtont létrehozunk, akkor az objektumok aktuális száma a címkéje elején kiíródik. Meghívhatjuk a GetTotal osztálymetódust az újonnan létrehozott objektum számára (figyeljük arra, hogy egy with kifejezésen belül vagyunk), csak úgy, mintha bármilyen más, megszokott metódust hívtunk volna, azzal a különbséggel, hogy meghívhatjuk ugyanezt a metódust érvényes objektum példány nélkül is. Ezt tesszük mi is, amikor az időzítő, amelyet hozzáadtunk, lejár:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Caption := Format ('CountObj: %d custom buttons',
    [TCountButton.GetTotal]);
end;
```

A Caption tulajdonság ebben a kódban a formra vonatkozik. Láthatjuk ennek a hívásnak az eredményét a 6.2. ábrán. Ennek a példának a hátulütője az, hogy

csak létrehozunk objektumokat, és nem semmisítjük meg őket, így mindig az összes élő objektum számát látjuk, amely mindig csak növekszik.



6.2. ábra *Néhány TCountButton objektum létrehozása a CountObj példában*

Az élő objektumok számának nullára csökkenését megnézhetjük a form, és a rajta lévő TCountButtonok megsemmisítése után. Ezt a kódot az egység végéhez adtam hozzá:

```
initialization
  MessageBox (0, PChar (Format (
    'There are %d CountButton objects',
    [TCountButton.GetTotal])),
    'Initialization', mb_OK);
finalization
  MessageBox (0, PChar (Format (
    'There are %d CountButton objects',
    [TCountButton.GetTotal])),
    'Finalization', mb_OK);
end.
```

Mindkét esetben a program egy egyszerű Windows üzenetablakot jelenített meg, amelyben a létező objektumok számát láthattuk, amit a GetTotal osztálymetódussal kaptunk meg. Ha lefuttatjuk a programot, akkor mind a kétszer nullát mutat az üzenetablakban. Hogy lássuk, hogy minden valóban működik, ugyanezt a kódot a form OnDestroy eseménykezelőjéhez is hozzáadtam, amely a form megsemmisítésekor hívódik meg.

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  MessageBox (0, PChar (Format (
    'There are %d CountButton objects',
    [TCountButton.GetTotal])),
    'Destroy', mb_OK);
end;
```

Ha most lefuttatjuk a programot, akkor látni fogjuk, hogy amikor a FormDestroy esemény elindul, akkor az összes létrehozott objektum még mindig létezik, de utána már megsemmisülnek, és a számláló értéke nullára csökken. Egy sokkal komplettebb példát is láthatunk, amelyben már futás alatt is megsemmisíthetjük a gombokat, miután tisztáztuk a metódus-mutatókat.

Metódus-mutatók

A metódus-mutatókat a Delphi adta az Object Pascal nyelvhez. A metódus-mutató típus olyan, mint egy eljárás típus, amely egy metódusra mutat. Gyakorlatilag egy ilyen mutatónak van egy rejtett Self paramétere. Más szavakkal, egy metódus-mutató két címet tárol: a metódus kódjának címét, és az objektum példány címét. Az objektum példány címe a metóduson belül Selfként jelenik meg, ez megmagyarázza a már említett TMethod típus definícióját, ami egy kód és egy adatmezőből álló rekord.

A metódus-mutatót hasonlóan kell deklarálni, mint az eljárás típusút, azzal a kivétellel, hogy a végén van egy kulcsszó:

```
type
  IntProceduralType = procedure (Num: Integer);
  IntMethodPointerType = procedure (Num: Integer)
    of object;
```

Ha deklarálunk egy olyan metódus-mutatót, mint ez, akkor egy objektumban lehet ilyen típusú mező:

```
type
  MyClass = class
    Value: Integer;
    Operation: IntMethodPointerType;
  end;
```

Később hozzárendelhetünk ehhez a mezőhöz bármilyen más típus-kompatíbilis metódust (eljárást vagy függvényt), ugyanolyan adattípusú paraméterekkel. Például deklarálhatunk egy másik osztályt, amelynek van egy metódusa, ugyanolyan egész paraméterrel:

```
type
  AnotherClass = class
    X: Integer;
    procedure Add (N: Integer);
  end;
```

Ha most deklarálunk egy pár objektumot mindkét osztályból, mint például:

```
var
  MyObject: MyClass;
  AnotherObject: AnotherClass;
```

Akkor végrehajthatjuk a hozzárendelést:

```
MyObject.Operation := AnotherObject.Add;
```

Most, ha meghívjuk a metódust:

```
MyObject.Operation;
```

Akkor az AnotherObject Add metódusát hívjuk meg.

Elsőre ennek a meglehetősen bonyolult technikának nem igazán érthető a célja, de ez a Delphi komponens technológiájának egyik alapja. Ha létrehoztunk egy objektumot, amelynek van néhány metódus-mutatója, akkor szabadon változtathatjuk a viselkedését, egyszerűen egy új metódus hozzárendelésével.

Amikor egy gombhoz hozzáadjuk az OnClick eseménykezelőt, akkor a Delphi pontosan ezt teszi. A gombnak lesz egy OnClick nevű metódus-mutatója, amelyhez közvetlen, vagy közvetett módon hozzárendelhetjük a form egyik metódusát. Amikor a felhasználó rákattint a gombra, akkor ez a metódus indul el, még akkor is, ha egy másik osztályban lett definiálva (tipikusan a formban).

A következő lista pontosan azt csinálja, amit az előző kódrészletek, az osztályok neveinek, metódusainak, és paramétereinek kis módosításával. Az új kód nagyon hasonlít a Delphi komponensei által használtra:

```
type
  TNotifyEvent = procedure (Sender: TObject) of object;

  MyButton = class
    OnClick: TNotifyEvent;
  end;

  TForm1 = class (TForm)
    procedure OnButton1Click (Sender: TObject);
    Button1: MyButton;
  end;

var
  Form1: TForm1;
```

Most, egy eljárás belsejébe beírhatjuk:

```
MyButton.OnClick := Form1.OnButton1Click;
```

Az igazi különbség a fenti kódrészlet és a VCL-ben található kód között az, hogy az OnClick egy tulajdonságnév, és az aktuális adatot, ami vele

kapcsolatos FOnClicknek hívják. Valójában az objektumvizsgáló (Object Inspector) események (Events) oldalán látható esemény nem több, mint egy metódus-mutató típus tulajdonsága.

A számlálós példa fejlesztése

Most, hogy már tudjuk, hogyan kell a metódus-mutatókat használni, továbbfejleszthetjük az előző példát. Az új példa neve CountObj2 lesz. Az általunk dinamikusan létrehozott új objektumokhoz adunk hozzá egy OnKeyPress eseménykezelőt. Ez valójában meglehetősen egyszerű: adjuk hozzá a következő kódot a form osztálydeklarációjához:

```
procedure ButtonKeyPress(Sender: TObject; var Key:
Char);
```

Az eseménynek szüksége van a felsorolt paraméterekre. Ha egy form komponensének OnKeyPress eseményét kiválasztva F1-et nyomunk, a Helpben a következőt találjuk:

```
TKeyPressEvent = procedure (Sender: TObject;
  var Key: Char) of object;
property OnKeyPress: TKeyPressEvent;
```

Ahogy azt látjuk, az esemény a TKeyPressEvent metódus-mutató típuson alapul. Ebből kifolyólag írnunk kell egy olyan metódust, amely eleget tesz ennek a metódus-mutató típusnak, mint amilyet feljebb láthattunk is.

Hogy kapcsolatba hozzuk ezt az eseményt a gombok OnKeyPress azon eseményeivel, amelyeket dinamikusan hozunk létre, egyetlen sort kell csak beírnunk a FormMouseDown metódusba:

```
with TCountButton.Create (Self) do
begin
  ...
  // set the event handler
  OnKeyPress := ButtonKeyPress;
  // grab the input focus
  SetFocus;
end;
```

A kód második sora az újonnan létrehozott gombra fókuszál (a fókuszálásról, és a billentyűzet kezeléséről általában, a 8. és 10. fejezetekben tudhatunk meg többet).

Most már megírhatjuk a ButtonKeyPress metódus valódi kódját. A Delphi nem hozza automatikusan létre a metódus vázát, ezért mindent nekünk kell megírnunk az egység implementációs részében, még az eljárás deklarációját is:

```
procedure TForm1.ButtonKeyPress(Sender: TObject; var
    Key: Char);
begin

end;
```

Természetesen ki is kell töltenünk ezt a metódus deklarációt valódi kódokkal. Ebben a példában azt akarjuk elérni, hogy ha a BACKSPACE billentyűt megnyomjuk, akkor megsemmisüljön a gomb. Mivel a billentyűzet inputja ahhoz a kontrollhoz kerül, amelyiken a fókusz van, ezért egyszerűen csak rá kell kattintanunk, vagy a TAB billentyűvel ki kell választanunk azt a gombot, amelyiket meg akarjuk semmisíteni, és meg kell nyomnunk a BACKSPACE-t.

Első megközelítésként megpróbáltuk megsemmisíteni a Sender paraméterként átadott objektumot, amely az eseményt fogadó objektum is egyben:

```
procedure TForm1.ButtonKeyPress(Sender: TObject;
    var Key: Char);
begin
  if Key = #8 then
    Sender.Free
end;
```

Ez a kód kivételt vált ki. Egyszerűen nem semmisíthetünk meg egy objektumot, amíg egy eseményét hajtjuk végre. Amit tehetünk az, hogy elmenthetjük a megsemmisítendő objektumot a form osztály egy privát mezőjébe, és később semmisítjük meg. Itt a kód, ami a példában is található:

```
procedure TForm1.ButtonKeyPress(Sender: TObject;
  var Key: Char);
begin
  // if user pressed backspace
  if Key = #8 then
    // set this as the object to destroy
    ToDestroy := Sender as TButton;
end;
```

Ebben a kódban a ToDestroy a form egy privát TButton típusú mezője:

```
private
  ToDestroy: TButton;
```

Ez a mező automatikusan NIL értéket kap (nincs megsemmisítendő objektum), amikor a formot először létrehozzuk (ez az alapértelmezés szerinti beállítás az osztálymezők részére). Amikor a felhasználó megnyomja a Backspace billentyűt, akkor az aktuális gomb objektum (a ButtonKeyPress metódus küldője) eltárolódik a ToDestroy mezőben. Most hozzáadhatunk a formhoz egy időzítő (Timer) komponenst, és minden alkalommal, amikor az adott időintervallum letelik, ellenőrizhetjük, hogy van-e megsemmisítendő gomb, és ha igen, akkor meg is semmisítjük:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  // if there is an object to destroy
  if Assigned (ToDestroy) then
  begin
    // moves the input focus to the next control
    SelectNext (ToDestroy, true, true);
    // destroy the object
    ToDestroy.Free;
    // no more objects to destroy
    ToDestroy := nil;
  end;
  // update the form caption
  Caption := Format ('CountObj: %d custom buttons',
    [TCountButton.GetTotal]);
end;
```

A program jobb működése érdekében, mielőtt megsemmisítünk egy objektumot, a fókuszt állítsuk a következő kontrolra, a SelectNext metódust meghívva. Ezután meghívjuk az objektum Free metódusát, amely meghívja nekünk a destruktort (megsemmisítőt). A destruktor virtuális, ezért a program a TButtonCount osztály fölülírt destruktorát hívja meg, amely csökkenti az objektum-számlálót, így tehát az objektum megsemmisítését a form címkéjének frissítése elé kell raknunk.

Objektumok egyszeri megsemmisítése

Ez a példa egy új témát hoz fel: minden létrehozott objektumot meg kell semmisíteni, de ha ugyanazt az objektumot kétszer próbáljuk meg megsemmisíteni, akkor az kivételt vált ki. Ezek sokoldalú problémák, és feltárásuk közben megbeszéljük a Free metódus, és a Destroy destruktor kapcsolatát, az Assigned függvény használatát egy objektum létezésének ellenőrzésére, és az objektumok NIL-re állításának okát. A biztonságos programok írásának alapja ezeknek a témáknak az alapos ismerete.

Először nézzük meg, hogy mi is a destruktor. A destruktor egy metódus, amely felszabadítja az objektum által lefoglalt memóriát. Írhatunk kódot a destruktorba, általában a Destroy destruktort fölülírva, hogy az objektum lefuttathasson még valamilyen kódot, mielőtt megsemmisül. A kódunkban természetesen nem kell a memória felszabadítással foglalkoznunk, ezt a Delphi megteszi helyettünk.

A Destroy a TObject osztály virtuális destruktora. A legtöbb osztálynak van egy saját virtuális „tisztogató" (clean-up) kódja, amely az objektum

megsemmisítésekor felülírja ezt a virtuális metódust. Azért nem szabad új konstruktort definiálnunk, mert az objektumok általában a Free metódus meghívásával semmisülnek meg, amely meghívja a Destroy virtuális destruktort (beleértve az általunk fölülírtat is).

Ahogy említettem a Free a TObject osztály egy metódusa, amelyet minden más osztály is örököl. A Free metódus gyakorlatilag ellenőrzi, hogy az aktuális objektum (Self) nem NIL-e, mielőtt meghívná a Destroy virtuális destruktort. Itt a kódja (az aktuális kód az RTL forráskód-fájlokban assemblerben van írva):

```
procedure TObject.Free;
begin
  if Self <> nil then
    Destroy;
end;
```

Most nézzük meg egy kicsit az Assigned funkciót. Amikor átadunk egy mutatót ennek a funkciónak, akkor ellenőrzi, hogy mutató NIL-e. A következő két kifejezés eredménye a legtöbb esetben megegyezik:

```
if Assigned (ToDestroy) then ...

if ToDestroy <> nil then ...
```

Ezek a kifejezések azt ellenőrzik, hogy a mutató nem NIL, azt azonban nem, hogy érvényesek-e. Ha a következő kódot írjuk:

```
ToDestroy.Free;
if ToDestroy <> nil then

  ToDestroy.DoSomething;
```

Akkor az ellenőrzés megfelelő lesz, és hibaüzenetet kapunk az objektum metódusának meghívásának sorában. Fontos, hogy felismerjük, hogy a Free nem állítja az objektumot NIL-re,

Egy objektumot nem állíthatunk automatikusan NIL-re: lehet több hivatkozásunk is ugyanarra az objektumra, és a Delphi ezt nem követi nyomon. Ugyanakkor egy metódusban (mint a Free) végezhetünk műveleteket az objektumon, de nem tudjuk a metódus hívására használt változó memória címét. Más szavakkal, a Free metóduson belül, vagy az osztály bármely metódusán belül, tudjuk az objektum memória címét (Self), de nem tudjuk a rá ToDestroyként hivatkozó változó memóriacímét. Ennél fogva a Free metódus nincs hatással a ToDestroy változóra.

Nézzük ezt a példát:

```
ToDestroy.Free;

...

ToDestroy.Free;
```

Természetesen valószínűleg nem fogunk ilyet írni, de előfordulhat, hogy egy sorban kétszer indítjuk el ugyanazt a metódust. Így fordíthatnánk ezt le:

```
if ToDestroy <> nil then

  ToDestroy.Destroy;

...

if ToDestroy <> nil then

  ToDestroy.Destroy;
```

Mivel a ToDestroy változó soha sem NIL, a Destroy metódus második meghívásakor is lefut, akkor is, ha az objektum már megsemmisült, és kivételt vált ki. Hogy ezt ellenőrizzük, töröljük ki a NIL-hez való hozzárendelést a CountObj2 példából, és egy ilyen hibát kapunk. Összegzésül néhány irányadó gondolat:

- Mindig a Free metódust hívjuk meg az objektumok megsemmisítésére.
- Mindig állítsuk NIL-re az objektum-hivatkozásokat a Free meghívása után, hacsak nem kerül azonnal hatáskörön kívülre a hivatkozás.

Osztályhivatkozások

Több, metódusokkal kapcsolatos téma átnézése után az osztályhivatkozásokkal foglalkozunk, még jobban kibővítve az objektumokat dinamikusan létrehozó példánkat. Az első fontos dolog az, hogy az osztályhivatkozás nem egy osztály, nem hivatkozás egy osztályra; egyszerűen csak hivatkozás egy osztály típusra.

Az osztályhivatkozás típus meghatározza egy osztályhivatkozás változójának típusát. Zavarosnak hangzik? Néhány programsor tisztábbá teheti a képet. Tételezzük fel, hogy definiáltunk egy osztályt:

```
type
  MyClass = class
    ...
  end;
```

Most definiálhatunk egy új osztályreferencia típust, ami ehhez az osztályhoz kapcsolódik:

```
type
  MyClassRef = class of MyClass;
```

Most deklarálhatunk mindkét típusból változókat:

```
var
  AClassRef: MyClassRef;
  AnObject: MyClass;
```

Az első változó egy objektumra hivatkozik, a második egy osztályra:

```
AClassRef := MyClass;
AnObject := MyClass.Create;
```

Vajon mire jó az osztályhivatkozás? Általában velük futás közben változtathatjuk meg egy osztály adattípusát. Bármilyen kifejezésben használhatunk osztályhivatkozásokat, ahol az adattípusok használata megengedett. Valójában nincs sok ilyen kifejezés, de az a néhány eset érdekes. A legegyszerűbb eset egy objektum létrehozása. Így is írhatjuk a fenti két sort:

```
AClassRef := MyClass;
AnObject := AClassRef.Create;
```

Itt a Create konstruktort az osztályhivatkozásnál használtuk, az aktuális osztály helyett, azaz egy osztályhivatkozást használtunk egy ilyen osztályú objektum létrehozására.

Megjegyzés

Az osztályhivatkozás a metaclass koncepciójára emlékeztet, amely más OOP nyelvekben megtalálható. Az Object Pascalban egy osztályhivatkozás nem egy osztály, csak egy típusmutató. Ennél fogva a metaclassokkal való hasonlatosság (osztályok, amelyek más osztályokat írnak le) egy kissé félrevezető. Gyakorlatilag a TMetaclass kifejezés használatos a Borland C++ Builderben.

Az osztályhivatkozások nem lennének olyan hasznosak, ha nem támogatnák ugyanazt a típus-kompatibilitási szabályt, amit az osztálytípusok. Amikor deklarálunk egy osztályhivatkozás változót, mint a MyClassRef fentebb, akkor

hozzárendelhetünk egy adott osztályt, és bármilyen alosztályt is. Tehát, ha a MyNewClass a MyClass alosztálya, akkor ezt is írhatjuk:

```
AClassRef := MyNewClass;
```

A Delphi sok osztályreferenciát deklarál a run-time könyvtárban és a VCL-ben, a következőket beleértve:

```
TClass = class of TObject;
ExceptClass = class of Exception;
TComponentClass = class of TComponent;
TControlClass = class of TControl;
TFormClass = class of TForm;
```

Különösen a TClass osztályhivatkozás típus használható bármilyen általunk írt osztályra való hivatkozásra, mivel minden osztály a TObjectből származik. Ehelyett a TFormClass hivatkozást használja a legtöbb Delphi projekt. Az Application objektum CreateForm metódusának is mint paraméter van szüksége a form osztályára, hogy létrehozhassa.

```
Application.CreateForm(TForm1, Form1);
```

Az első paraméter egy osztályhivatkozás, a második az aktuális objektum.

Objektumok létrehozása futási idő alatt osztályhivatkozások használatával

Mi a gyakorlati jelentősége az osztályhivatkozások használatának? Az adattípusok futási idő alatti változtathatósága a Delphi alapvető eleme. Amikor a komponens palettából egy új komponenst adunk egy formhoz, akkor valójában egy adattípust választottunk ki, és egy ilyen típusú objektumot hoztunk létre, legalábbis a Delphi ezt teszi a háttérben.

A ClassRef példával jobban megérthetjük az osztályhivatkozások működését. A példában található form meglehetősen egyszerű. A form felső részében egy panelen belül található három rádiógomb. Amikor kiválasztjuk az egyiket, és a formra kattintunk, akkor a rádiógombok címkéjével jelölt típusú komponenseket hozhatunk létre (rádiógomb, nyomógomb, szerkesztő doboz).

A program megfelelő működéséhez meg kell változtatnunk a három komponens nevét. A formnak kell, hogy legyen egy osztályhivatkozás mezője is. Először deklarálnunk kell egy új osztályhivatkozás típust:

```
type
  TControlClass = class of TControl;
```

Valójában ez csak egy egyszerű újradeklarálás, mivel ez az osztályhivatkozás típus már megvan a VCL-ben. A következő lépésben két új, privát mezőt kell a formhoz adnunk:

```
private
  ClassRef: TControlClass;
  Counter: Integer;
```

Az első mező tárolja az új adattípust, ha a felhasználó a rádiógombokra kattint. A három metódusból itt van az egyik:

```
procedure TForm1.RadioButtonRadioClick(Sender: TObject);
begin
  ClassRef := TRadioButton;
end;
```

A másik két rádiógomb OnClick eseménye hasonló ehhez, csak a TEditet, vagy a TButtont rendelik a ClassRef mezőhöz. A form OnCreate eseményében is hasonló hozzárendelés van, amelyet inicializáló metódusként használ.

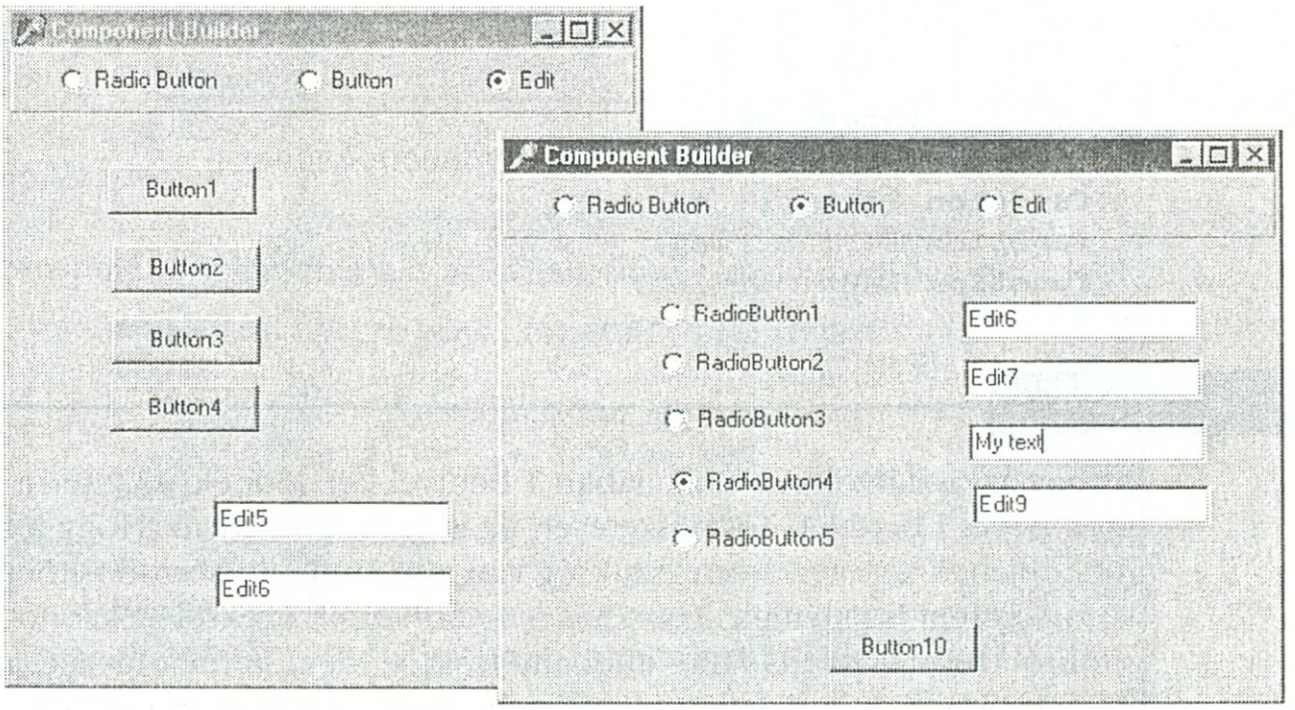
A kód érdekes része az, amikor a felhasználó újra a formra kattint. A form OnButtonDown eseményét használjuk az egérkattintás helyének meghatározására:

```
procedure TForm1.FormMouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  NewCtrl: TControl;
  MyName: String;
begin
  // create the control
  NewCtrl := ClassRef.Create (Self);
  // hide it temporarily, to avoid flickering
  NewCtrl.Visible := false;
  // set parent and position
  NewCtrl.Parent := Self;
  NewCtrl.Left := X;
  NewCtrl.Top := Y;
  // compute the unique name (and caption)
  Inc (Counter);
  MyName := ClassRef.ClassName + IntToStr (Counter);
  Delete (MyName, 1, 1);
  NewCtrl.Name := MyName;
  // now show it
  NewCtrl.Visible := true;
end;
```

A kód első sora a metódus kulcsa. Ez egy új, a ClassRef-ben tárolt, osztály adattípusú objektumot hoz létre. Ezt egyszerűen az osztályhivatkozás Create konstruktorának meghívásával hajtjuk végre. Most már beállíthatjuk az új komponens Parent tulajdonságát, pozícióját, nevét (amelyet a címkéje is használ), és láthatóvá tehetjük.

Figyeljük meg, hogy a kód hozta létre a nevet, a Delphi névadási szabályait utánozva. Az osztály nevét a ClassRef.ClassName kifejezéssel tudtuk meg (ezt a TObject osztálymetódust a következő fejezetben beszéljük meg). A név végéhez hozzáraktunk egy számot, és a karakterlánc kezdőbetűjét levágtuk. Például az első rádiógomb alapértelmezés szerinti elnevezése TRadioButton, a végéhez hozzáírunk egy „1"-est, és az elejéről levágjuk a „T" betűt (RadioButton1).

A 6.3. ábrán láthatunk példát a program két kimenetére. Az elnevezések nem pontosan olyanok, mint amit a Delphi is használ. A Delphi minden típusú kontrolnál külön számlálót használ; a példában az összes komponens egyet. Ha a ClassRef példaprogramban a formra helyezünk egy rádiógombot, egy nyomógombot, és egy szerkesztőt, akkor az elnevezésük RadioButton1, Button2, és Edit3 lesz.



6.3. ábra *A ClassRef két példája két különböző ablakban*

Interfész és többszörös öröklődés

Az előző fejezetben láthattuk, hogy a Delphi nem támogatja a többszörös öröklődést: minden osztálynak csak egyetlen szülő osztálya lehet. Láthattuk azt is, hogy létrehozhatunk absztrakt osztályt is, amelynek egy vagy több absztrakt metódusa van. A Delphi 3 újabb lehetőségeket is nyújt mindezek mellett.

Megjegyzés

Az ebben a fejezetben tárgyalt technikákat a COM objektumok implementálására használják, és ezekről részletesebben a 21-24. fejezetekben lesz szó. Egyelőre tekintsük őket nyelvi elemeknek.

Interfész deklarálása

Az absztrakt osztályok deklarálása mellett (amelyeknek absztrakt metódusai vannak), a Delphi 3-ban teljesen absztrakt osztályt is deklarálhatunk, amelynek csak virtuális absztrakt metódusai vannak. Ezt az interface kulcsszóval valósíthatjuk meg. Ezért az ilyen osztály deklarációkra, mint interfészre hivatkozunk. Technikailag egy interfész nem egy osztály, ámbár hasonlíthat rá. Itt van egy új interfész definíciója:

```
type
  ICanFly = interface
    ['{10000000-0000-0000-0000-000000000000}']
    function TakeOff: string;
    function Fly: string;
    function Land: string;
  end;
```

Tipp

Ahogy az osztályok nevei általában T betűvel kezdődnek, az interfészek nevei I betűvel. Természetesen ez is csak konvenció. Hogy jól működjenek az interfészek, szükség van egy numerikus azonosítóra is, ahogy fent is láthatjuk. Többet is megtudhatunk ezekről az azonosítókról (hivatalosan GUID – Globally Unique Identifiernek hívják) a 21. fejezetben. Itt majdnem teljesen tetszőleges értéket használtunk.

Ez az interfész nem igazán hasznos addig, amíg nem kezdünk el belőle valódi osztályokat származtatni. Egy osztálynak, amelyet egy interfészből örökítettünk, implementálnia kell az őse összes metódusát, lehetőleg virtuálisként deklarálva:

```
type
  TAirplane = class (TInterfacedObject, ICanFly)
    function TakeOff: string; virtual;
    function Fly: string; virtual;
    function Land: string; virtual;
    destructor Destroy; override;
  end;
```

Ahogy minden osztály a TObject osztályból öröklődik, úgy minden interfész az IUnknown interfészből (erről az interfészről többet a 21. fejezetben tudhatunk meg). Mivel az ICanFly az IUnknownból öröklődik, a TAirplane osztálynak implementálnia kell mindkét interfész metódusait. Delphiben van egy különleges osztály, a TInterfacedObject, ami az IUnknown interfész alapvető metódusait implementálja. Ezért a TAirplane osztályt a TInterfacedObjectből is örökítettük.

Figyelem!

Valójában a fordító nem fogad el olyan osztályt, amely csak egy interfészből öröklődik. Ezt megkerülhetjük, ha az interfészből, és a TObject osztályból örökítünk, például:

```
TAirplane = class (TObject, ICanFly).
```

Most már implementálhatjuk a TAirplane osztály metódusait:

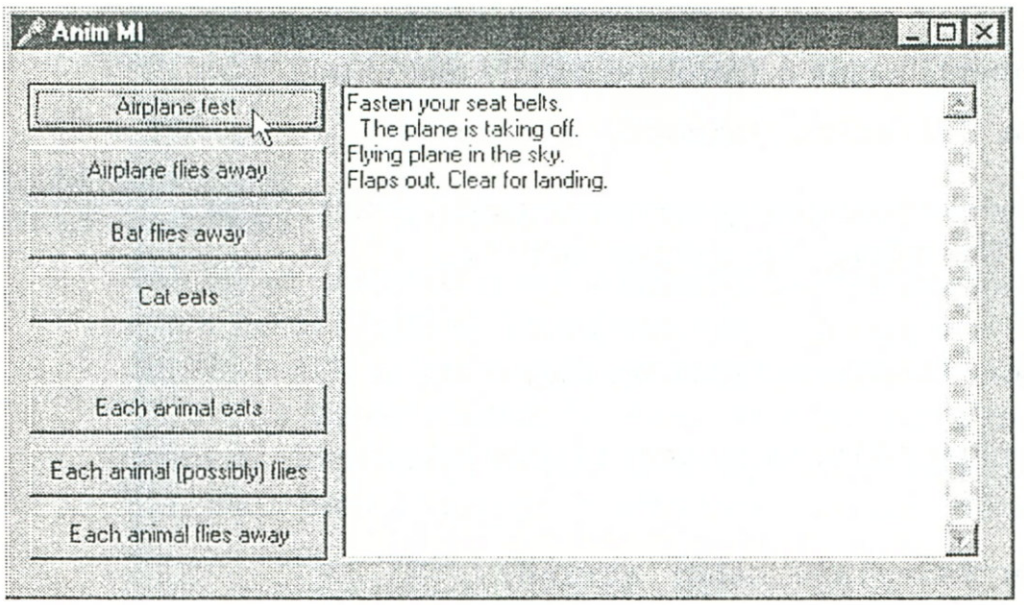
```
function TAirplane.TakeOff: string;
begin
  Result := 'Fasten your seat belts. '#13#10+
    '  The plane is taking off.';
end;

function TAirplane.Fly: string;
begin
  Result := 'Flying plane in the sky.';
end;

function TAirplane.Land: string;
begin
  Result := 'Flaps out. Clear for landing.';
end;

destructor TAirplane.Destroy;
begin
  MessageBox (0, 'Airplane destroyed',
    'Anim MI', mb_ok);
end;
```

Ezt a destruktort azért adtuk hozzá, hogy lássuk, ha az osztály egyik objektuma megsemmisül. A Destroy metódus a MessageBox API funkciót használja, így a program futtatásának legvégén láthatjuk az üzenetet, ahogy azt korábban a

CountObj példában is csináltuk. Ha egyszer elkészítettük, akkor írhatunk egy példaprogramot, ami úgy használja ezt az osztályt, akárcsak a többit, valójában nincs köztük különbség:

```
procedure TForm1.ButtonAirClick(Sender: TObject);
var
  Air1: TAirplane;
begin
  Air1 := TAirplane.Create;
  Memo1.Lines.Add (Air1.TakeOff);
  Memo1.Lines.Add (Air1.Fly);
  Memo1.Lines.Add (Air1.Land);
  Memo1.Lines.Add ('');
  Air1.Free;
end;
```

A 6.4. ábrán láthatjuk ennek a kódnak az eredményét, ami a bal első gomb megnyomásakor történik. Az AnimMI példa többi gombjáról még szó lesz a fejezetben.



6.4. ábra

Az AnimMI példa Airplane teszt gombjának megnyomásának hatása

Az utolsó kódrészletben deklaráltunk egy TAirplane osztályú objektumot, és használtuk is. Ha egyszer deklaráltunk egy új interfész típust, akkor deklarálhatunk egy olyan típusú változót is. Egy ilyen változó hasonlít egy osztály típusú változóra, hozzárendelhetünk olyan osztályú objektumokat is, amelyek implementálják az interfészt. Például:

```
procedure TForm1.ButtonAirIntClick(Sender: TObject);
var
  Fly1: ICanFly; // interface
begin
  Fly1 := TAirplane.Create;
  Memo1.Lines.Add (Fly1.TakeOff);
  Memo1.Lines.Add (Fly1.Fly);
  Memo1.Lines.Add (Fly1.Land);
  Memo1.Lines.Add ('');
end;
```

Figyeljük meg, hogy nincs olyan programsor, ami megsemmisítené az általunk létrehozott objektumot. Ha lefuttatjuk ezt a kódot (a form második gombját megnyomva), akkor ugyanazt az eredményt kapjuk, mint az előző módszernél, a destruktor meghívásával. Ezt bizonyítja az üzenet ablak megjelenése is.

Amint hozzárendelünk egy interfészváltozóhoz egy objektumot, a Delphi automatikusan ellenőrzi, hogy az objektum implementálja-e az interfészt, és ha igen, akkor meghívja az _AddRef metódust, ami megnöveli az interfészváltozó hivatkozásszámlálóját. A metódus végén, ha az interfészváltozót már többet nem használjuk, a Delphi meghívja a _Release metódust, ami csökkenti a hivatkozásszámlálót, ellenőrzi, hogy az értéke egyenlő-e nullával, mert abban az esetben megsemmisíti az objektumot.

Más szavakkal, a Delphi 3 az interfészváltozók objektumokra való hivatkozásait nyilvántartja, és az objektumok automatikusan megsemmisülnek, ha már nem hivatkozik rájuk egyetlen interfészváltozó sem.

Figyelem!

Ez nem veszi figyelembe, hogy esetleg egy objektumváltozó még hivatkozhat az objektumra, ezért soha ne keverjük össze az objektumokat és a hivatkozásokat!

Interfész használata paraméterként

Érdekes, hogy bizonyos szintig az interfészeket is lehet polimorfálni. Valójában írhatunk olyan funkciót, amely interfészen végez műveleteket, mint például a következő FlyAway funkció:

```
function FlyAway (F: ICanFly): string;
begin
  Result := F.TakeOff + #13#10 + F.Fly + #13#10 +
  F.Fly + ' ...over and over... ' + #13#10 +  F.Land;
end;
```

A fenti kódban a #13#10 karakterek a memo komponensben új sort adnak. Ugyanazt az eredményt kapjuk, mint ami a fenti ButtonAirClick metódus kimenete a következő sorok begépelésével is:

```
procedure TForm1.ButtonAirFAClick(Sender: TObject);
var
  Air1: ICanFly;
begin
  Air1 := TAirplane.Create;
  Memo1.Lines.Add (FlyAway (Air1));
  Memo1.Lines.Add ('');
end;
```

Figyelem

Amikor paraméterként adunk át egy interfészt, akkor a Delphi automatikusan növeli, és utána csökkenti a hivatkozás számlálóját. Ha nem megfelelően inicializáljuk a hivatkozásszámlálót (például, ha deklarálunk egy TAirplane típusú változót), akkor automatikusan megsemmisülhet, ha a meghívott funkcióból visszatér a program, az objektumot érvénytelenné téve a következő sorokban.

Egy második interfész írása

Az összes fent megbeszélt kód az AnimMI példában van. A név jelentése Állatok Többszörös Öröklődése (Animals Multiple Inheritance), és ámbár egy repülőgépnek semmi köze az állatokhoz, újra az ICanFly interfészt fogjuk használni, és néhány más funkciót, a FlyAway-t is beleértve. A fejezet ezen részének összes kódját egyetlen példába fogjuk sűríteni, az osztályokat, és a formokat külön egységbe írjuk. A formnak, ahogy azt a 6.4. ábrán is láthattuk, van egy memo komponense a kiíratásra, és vannak gombjai egy csoportban, amelyek címkéi jelzik működésüket.

Az „állatokhoz" hozzáadunk egy másik interfészt is, utána írunk egy egyszerű osztályt az „emlősök" (mammals) definiálására:

```
type
  IMammal = interface
    ['{20000000-0000-0000-0000-000000000000}']
    function Eat: string;
  end;
```

```
TMammal = class (TInterfacedObject, IMammal)
  function Eat: string; virtual; abstract;
  destructor Destroy; override;
end;
```

A definíció szerint az emlősök tudnak enni, ez az Eat (természetesen ezt egyéb állatokról is el lehet mondani. Más módon is megmagyarázhattuk volna, de egy példa lehetőség szerint legyen egyszerű, nem igaz?). A fenti osztály szintén a TInterfacedObjectből öröklődik. Hogy az interfésszel kapcsolatos technikákat használhassunk, mint például az interfész-lekérdezés, a TMammal osztályban implementálnunk kell legalább egyet. Ez a TMammal osztály destruktora lesz, amelyet az összes alosztálya is használni fog:

```
destructor TMammal.Destroy;
var
  S: String;
begin
  S := ClassName + ' object gone';
  MessageBox (0, PChar (S), 'TMammal.Destroy', mb_OK);
end;
```

A TMammalból örökítettük a TCat osztályt a következő kóddal:

```
type
  TCat = class (TMammal)
    function Eat: string; override;
  end;

function TCat.Eat: string;
begin
  Result := 'The cat is eating a (Microsoft) mouse.';
end;
```

Ismét egyszerűen használhatunk ilyen osztályú objektumokat, és meghívhatjuk a metódusaikat is:

```
procedure TForm1.ButtonCatClick(Sender: TObject);
var
  Cat1: IMammal;
begin
  Cat1 := TCat.Create;
  Memo1.Lines.Add (Cat1.Eat);
  Memo1.Lines.Add ('');
end;
```

Ismételten, mivel egy interfész típusú változót használtunk egy sima osztály típusú változó helyett, nem kell megsemmisítenünk az objektumot manuálisan.

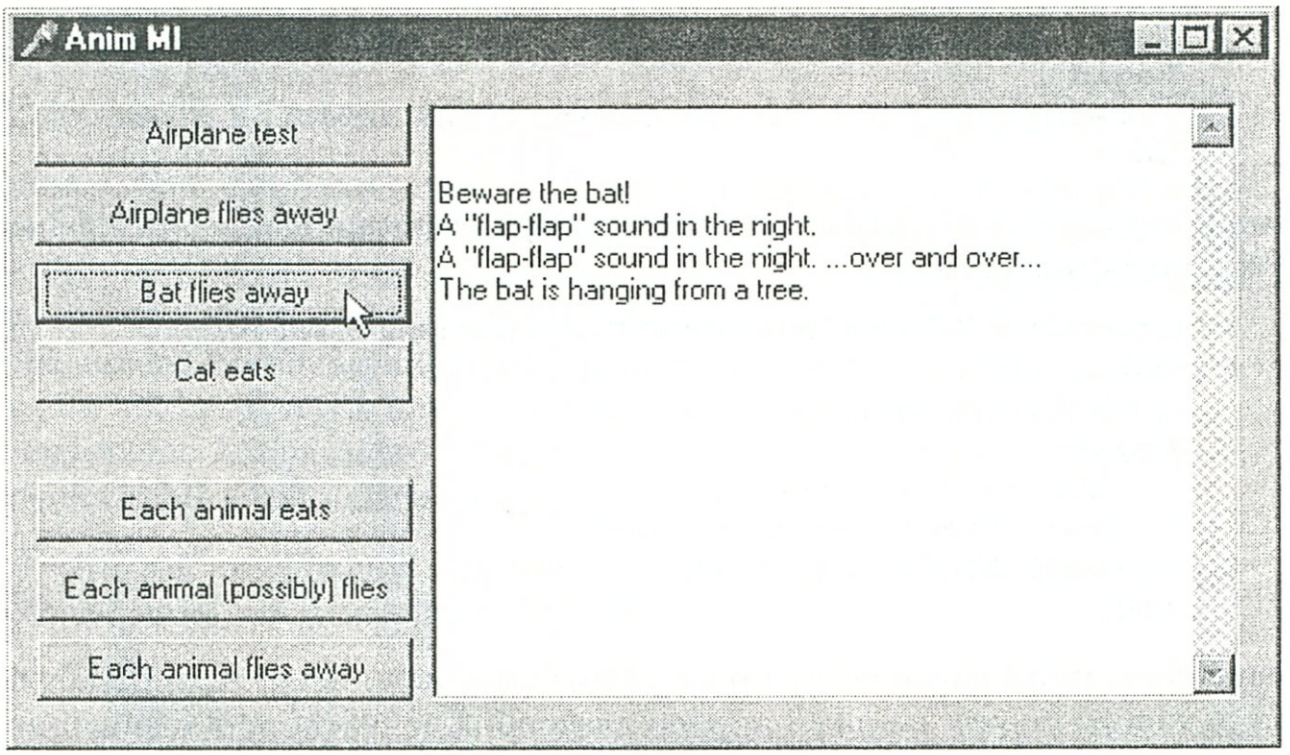
Mindkét interfész implementációja

Most jön az érdekes rész. Vehetünk egy emlőst, amely implementálja az ICanFly interfészt is, mint például egy denevér (bat):

```
type
  TBat = class (TMammal, ICanFly)
    function Eat: string; override;
    function TakeOff: string; virtual;
    function Fly: string; virtual;
    function Land: string; virtual;
  end;
```

Ismét három metódusát implementáltuk az osztálynak, és most ugyanúgy használhatjuk őket, mint a ButtonCatClick metódusban, az előző rész végén. A következő kód meghívja a FlyAway funkciót, ami az ICanFly interfészen dolgozik (a 6.5. ábrán láthatjuk a kód eredményét):

```
procedure TForm1.ButtonBatFAClick(Sender: TObject);
var
  FlAnim: ICanFly;
begin
  FlAnim := TBat.Create;
  Memo1.Lines.Add (FlyAway (FlAnim));
  Memo1.Lines.Add ('');
end;
```



6.5. ábra *Ez történik az AnimMI példában egy denevér (bat) FlyAway funkciójának meghívásakor*

Ebben a metódusban ismét egy interfész típusú változót használtunk. Igaz, hogy használhattunk volna IMammal típusú változót, vagy egyet az ICanFly típusból, mivel a TBat osztály mindkét interfészt implementálja, a fenti metódust írhatjuk így is:

```
procedure TForm1.ButtonBatFAClick(Sender: TObject);
var
  FlAnim: IMammal;
begin
  FlAnim := TBat.Create;
  Memo1.Lines.Add (FlyAway (FlAnim as ICanFly));
  Memo1.Lines.Add ('');
end;
```

Ez a kód az as operátort használja, hogy az objektum más interfészt használjon, jelen esetben az ICanFly-t. Az as operátornak, amit itt az interfészek használnak, nincs köze az RTTI as operátorához, az objektumok és az osztályok használják. Amikor az as operátor második operandusa egy interfész, a Delphi meghívja a QueryInterface-t, ami az összes interfész által ismert speciális metódus, hogy ellenőrizze, hogy az interfész támogatva van-e, és ha igen, akkor átkonvertálja az interfész objektumot az új értékre. Valójában ez történik akkor is, ha egy objektumot rendelünk hozzá egy interfész-változóhoz, ahogy az előző metódus első sorában, amelyet így is írhatunk:

```
FlAnim := TBat.Create as IMammal;
```

Ez a példa demonstrálja, hogy egyetlen funkciónk is lehet, mint például a FlyAway, amely működik minden olyan objektumon, ami implementálja az ICanFly interfészt, legyen az denevér vagy repülőgép. Elértünk egy polimorfizmushoz hasonló dolgot, még akkor is, ha a TBat és a TAirplane nem egymásból öröklődtek. Mondhatjuk, hogy mindkettő implementálja ugyanazt az interfészt, ami ahhoz hasonló (de nem ugyanaz), mintha ugyanabból az absztrakt alaposztályból örökítettük volna őket.

A példa szépsége az, hogy a TMammal osztály, és annak leszármazottai is polimorfizálhatók. Az AnimMI példa fő formjának van egy állatokat tartalmazó tömbje:

```
private
    Animals: array [1..3] of IMammal;
```

Ezek az objektumok a program indulásakor jönnek létre, a form OnCreate eseményében. Ez a metódus aktiválja a hivatkozásszámlálást, és egy szöveget is kiír, ami leírja a műveletet:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Animals[1] := TCat.Create;
  Animals[2] := TBat.Create;
```

```
    Animals[3] := TCat.Create;
end;
```

Amikor a Form1 megsemmisül, akkor a három állat is automatikusan megsemmisül. Ezt a viselkedést a TMammal destruktora által megjelenített üzenetablak is megerősíti. Amikor az utolsó három gomb közül valamelyiket megnyomjuk, akkor műveleteket hajthatunk végre a tömb összes állatán. Az első gomb, az „Összes állat eszik" (Each animal eats) címkéjű is a polimorfizmus működésének példáját mutatja:

```
procedure TForm1.ButtonAllEatClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 3 do
    Memo1.Lines.Add (Animals[I].Eat);
  Memo1.Lines.Add ('');
end;
```

Ha meg akarjuk hívni a Fly, vagy FlyAway metódust az összes objektumra, amely az ICanFly interfészt is implementálja, akkor egy kis nehézségbe botlunk. Nem hívhatjuk meg közvetlenül a Fly metódust, és ha az as konverziót használjuk, akkor lehet, hogy az nem sikerül, és kivételt váltunk ki az „interfész nincs támogatva" (Interface not supported) üzenettel.

Ellenőriznünk kell, hogy az interfész támogatva van-e, mielőtt használnánk az as operátort. Ezt a QueryInterface metódust meghívásával tehetjük meg. Ez a metódus paraméterként várja a konvertálandó interfész objektumunkat, és egy másikat, amely tartalmazni fogja az eredményt. A metódus meghívása után az eredményt használhatjuk egy újabb konverzió helyett:

```
procedure TForm1.ButtonAllFlyClick(Sender: TObject);
var
  I: Integer;
  Fly1: ICanFly;
begin
  for I := 1 to 3 do
  begin
    Animals[i].QueryInterface (ICanFly, Fly1);
    if Assigned (Fly1) then
      Memo1.Lines.Add (Fly1.Fly);
  end;
  Memo1.Lines.Add ('');
end;
```

A QueryInterface metódus által visszaadott interfész paraméter értékének ellenőrzése helyett az Assigned (Fly1) meghívásával a metódushívás végeredményét ellenőrizhetjük. A QueryInterface nullát ad vissza, ha megtalálta az interfészt, ellenkező esetben egy hibakódot (E_NoInterface). Mivel a metódus

visszatérési értékének nullához hasonlításával való ellenőrzés furcsának tűnhet, az utolsó gomb metódusánál azt ellenőrizzük, hogy az eredmény E_NoInterface-től különböző-e. A ButtonAllFAClick metódus egy kissé más megközelítést használ, mint az előbb a ButtonAllFlyClick metódus:

```
procedure TForm1.ButtonAllFAClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 3 do
    if Animals[i].QueryInterface (
        ICanFly, Fly1) <> E_NoInterface then
      Memo1.Lines.Add (FlyAway(Fly1));
  Memo1.Lines.Add ('');
end;
```

Mostanra már láthatjuk, hogy ez egy nagyon érdekes kibővítése az Object Pascal nyelvnek. Nagyon hasonlót találhatunk más OOP nyelvekben is, különösen a Javában. Az interfészek implementációja a Javában kicsit rugalmasabb, amíg a Delphiben az implementáció szigorúan a COM objektumokhoz kötött. Ezért a 21. fejezetben innen folytatjuk.

Megjegyzés

A Javának és az Object Pascalnak több a közös vonása, mint azt a legtöbben gondolnák. A legtöbb programozó úgy gondolja, hogy a Java a C++-hoz hasonló, és ha csak a szintaxist nézzük, akkor igaz is. Ha az objektumhivatkozási modellre gondolunk, az egyszeri öröklési modellre, a kivétel kezelés módjára, és az interfészek szerepére, láthatjuk, hogy a Java és az Object Pascal között sokkal több a hasonlóság, mint köztük és a C++ között. Valójában mindkettő korábbi OOP nyelvekből vett szolgáltatásokat tartalmaz, a kevéssé ismert Eiffelt is beleértve.

Tulajdonságok definiálása

Most, hogy beszéltünk az interfészekről, a fejezet hátralévő részében az Object Pascal nyelv más kiterjesztéseivel is foglalkozhatunk, különösen a vizuális, komponens alapú programozással. Ebben a részben a tulajdonságokkal foglalkozunk, később az eseményekkel, és felépítünk egy egyszerű komponenst.

A tulajdonságok határozzák meg egy objektum állapotát és viselkedését. A tulajdonság alapvetően egy név, amely alapján közvetlenül elérhetünk adatokat, vagy a metódusok írhatják, olvashatják az értékét. Más szavakkal, minden

alkalommal, amikor megváltoztatjuk, vagy olvassuk egy tulajdonság értékét, akkor vagy egy mezőt érünk el (akár egy privátot is), vagy egy metódust hívunk meg. Itt van például a dátum objektum tulajdonságának definíciója:

```
property Month: Integer read FMonth write SetMonth;
```

Hogy elérjük a Month tulajdonság értékét, ennek a kódnak ki kell olvasnia az FMonth privát mező értékét, míg az érték megváltoztatásához a SetMonth metódust hívjuk meg. Különböző kombinációk lehetségesek (például használhatnánk egy metódust az érték kiolvasására, vagy közvetlenül egy mezőt írhatnánk a `write` direktíva után), de metódusok használata egy tulajdonság értékének megváltoztatására nagyon elterjedt módszer. Felsorolok néhány alternatívát:

```
property Month: Integer read GetMonth write SetMonth;

property Month: Integer read FMonth write FMonth;
```

Tipp

Amikor egy tulajdonságot akarunk elérni a programban, akkor fontos figyelnünk arra, hogy esetleg egy metódust kell meghívnunk, mert néhány metódus futása „sok" időt vehet igénybe. Mellékhatásaik is lehetnek, beleértve a komponens (lassú) újrarajzolását is a képernyőn. Ámbár a tulajdonságok mellékhatásai nem igazán vannak dokumentálva, létezésükkel tisztában kell lennünk, különösen, ha a kódot optimalizáljuk.

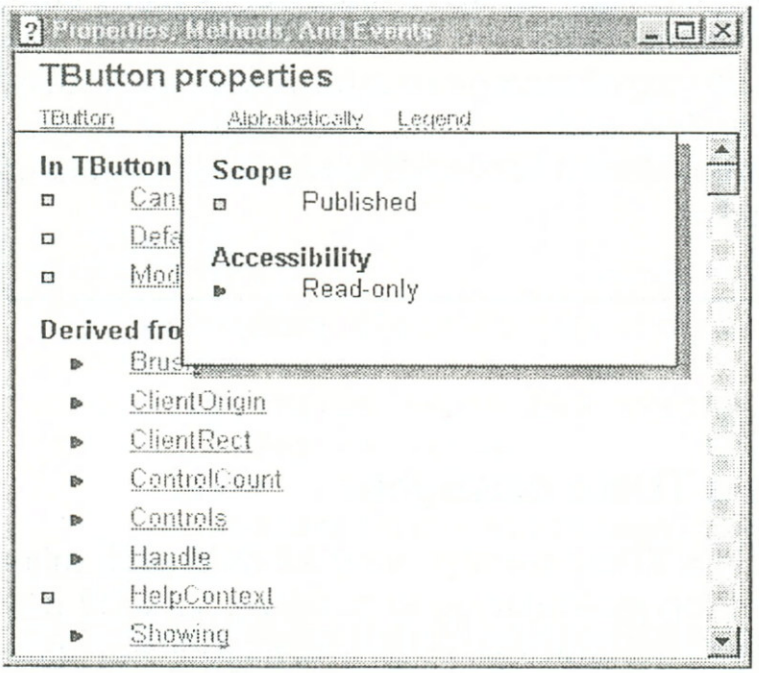
A tulajdonság `write` direktívája el is hagyható, csak olvashatóvá téve azt. Technikailag a `read` direktívát is elhagyhatjuk, és egy csak írható tulajdonságot definiálhatunk, de ez ritkán fordul elő. Másik különbség a szerkesztés alatti, és a csak futási idő alatti tulajdonságok között van. A szerkesztési idő alatti tulajdonságok az osztály deklaráció published részében definiálhatóak. Minden más, ami a public részben van deklarálva csak futási idő alatt érhető el. Az összes csak olvasható tulajdonságot a public részben kell definiálni (vagy a protected, vagy a private részekben), mert a published tulajdonságoknak írhatónak és olvashatónak kell lenniük.

A published tulajdonságok értékét szerkesztési idő alatt az objektumvizsgálóban (Object Inspectorban) írhatjuk, olvashatjuk. A Delphi vizuális programozói környezete ilyen módon teszi lehetővé a tulajdonságok elérését. Futási idő alatt mind a published, mind a public tulajdonságokat elérhetjük.

Megjegyzés

Ne felejtsük el, hogy az Object Inspector a komponensnek csak a tervezési idő alatti tulajdonságait listázza ki, a futási idejű tulajdonságokat kihagyva. A komponens tulajdonságainak teljes listáját megtalálhatjuk a Delphi Help fájljaiban.

Összességében, az objektumvizsgálóban kilistázott tulajdonságok mellett (szerkesztés alatt), vannak más tulajdonságok is (csak futási idejűek), amelyek közül néhány csak olvasható. A Help fájlokban megtalálhatjuk minden tulajdonság leírását. A Delphi Help rendszerében a csak olvasható, és a published tulajdonságokat szimbólumok jelzik, amelyet a 6.6. ábrán is láthatunk. A Delphi 2-ben ezek más szimbólumok voltak. Valójában az egész Súgó fájl struktúrája megváltozott, hogy könnyebben keresgélhessünk egy komponens osztály tulajdonságainak listájában, és annak alaposztályaiban.



6.6. ábra *Az új Delphi 3 Súgó fájlokban különböző szimbólumok jelzik a különféle típusú tulajdonságokat*

Általában hozzárendelhetünk értékeket egy tulajdonsághoz, vagy olvashatjuk, akár kifejezésekben is használhatjuk, de nem mindig adhatjuk át őket paraméterként eljárásoknak, vagy metódusoknak. Azért nem lehet, mert egy tulajdonság nem egy memóriahely, nem használható úgy, mint egy változó paraméter, nem adható át hivatkozással sem.

Nincs minden VCL osztálynak tulajdonsága. Komponenseknek, és a TPersistent osztály alosztályainak vannak tulajdonságai, mert általában fájlba menthetjük őket. A DFM fájl nem más, mint a formon található komponensek published tulajdonságainak összessége. A Delphi széleskörűen támogatja az ilyen jellegű információk fájlba mentését, ahogy azt a 29. fejezetben is láthatjuk majd.

Megjegyzés

A tulajdonság egy nagyon jó OOP mechanizmus, az egymásba ágyazás ötletének jól kigondolt megvalósítása. Alapvetően egy név rejti el az osztályinformációk elérésének implementációját (akár közvetlenül, akár metódushíváson keresztül érjük el az adatot). Valójában a tulajdonságok használatával végül eljutunk az interfészhez, amely nem valószínű, hogy megváltozik. Ugyanakkor, ha azt akarjuk, hogy a felhasználó csak néhány mezőt érjen el, akkor egyszerűbb tulajdonságok közé tenni őket a public változók használata helyett. Nem kell további kódot írnunk (az egyszerű Get és Set metódusok írása nagyon unalmas), és még mindig megváltoztathatjuk az osztály implementációját. Ha a közvetlen adatelérést metódusokon keresztülire cseréljük, akkor sem kell megváltoztatnunk a forráskódban a tulajdonságokat használó sorokat, csak újra kell fordítanunk a programot. Az egységbe zárás maximális kihasználásának lehetőségeként ez mindig jusson eszünkbe!

Tulajdonságok hozzáadása a TDate osztályhoz

Az előző fejezetben fejlesztettük ki a TDate osztályt, most kibővíthetjük tulajdonságokkal. Az új példa, a DateProp gyakorlatilag az 5. fejezet ViewD2 példájának kiterjesztése. Ez az osztály új deklarációja, néhány új metódussal (a tulajdonságok beállítására), és négy tulajdonsággal:

```
type
  TDate = class
  private
    FMonth, FDay, FYear: Integer;
  protected
    function DaysInMonth: Integer;
    procedure SetMonth (Value: Integer);
    procedure SetDay (Value: Integer);
```

```
  public
    constructor Init (m, d, y: Integer);
    procedure SetValue (m, d, y: Integer);
    function LeapYear: Boolean;
    procedure Increase;
    procedure Decrease;
    procedure Add (NumberOfDays: Integer);
    procedure Subtract (NumberOfDays: Integer);
    function GetText: string;
    // properties:
    property Day: Integer read FDay write SetDay;
    property Month: Integer read FMonth write SetMonth;
    property Year: Integer read FYear write FYear;
    property Text: string read GetText;
  end;
```

Az általános jelölést követve, a lokális mezők neve elé egy F betűt írtunk. A Day és a Month tulajdonságok közvetlenül olvassák, és a két megfelelő metódussal tárolják az értékeket. Azért adtuk hozzá ezt a két metódust, hogy ellenőrizzék, hogy a két érték megfelelő intervallumba essen:

```
procedure TDate.SetMonth (Value: Integer);
begin
  if (Value >= 1) and (Value <= 12) then
    FMonth := Value
  else
    raise EDateOutOfRange.Create ('Month out of
        range');
end;

procedure TDate.SetDay (Value: Integer);
begin
  if (Value >= 1) and (Value <= DaysInMonth) then
    FDay := Value
  else
    raise EDateOutOfRange.Create ('Day out of range');
end;
```

A kód megírásakor definiáltunk egy saját kivételkezelő osztályt, amely kiváltódik, ha az érték egy adott intervallumon kívülre esik:

```
type
  EDateOutOfRange = class (Exception);
```

A Year tulajdonság egyszerűen olvassa és írja a megfelelő privát mezőt. Az értéket ellenőrző metódusok megírása után frissíthetjük a SetValue metódust és az Init konstruktort, hogy használjuk is őket:

```
procedure TDate.SetValue (m, d, y: Integer);
begin
```

```
    FYear := y;
    // check the ranges
    SetMonth (m);
    SetDay (d);
end;
```

A negyedik tulajdonság a Text, csak egy metódushoz tartozik. Ennek a csak olvasható tulajdonságnak az értéke nincs tárolva az objektumban, minden alkalommal kiszámoljuk egy függvénnyel, ami már benne van az osztályban (valójában már használtuk a hónap szöveges leírását, ami a ViewD2 példaprogram örökített osztályának a része).

Tipp

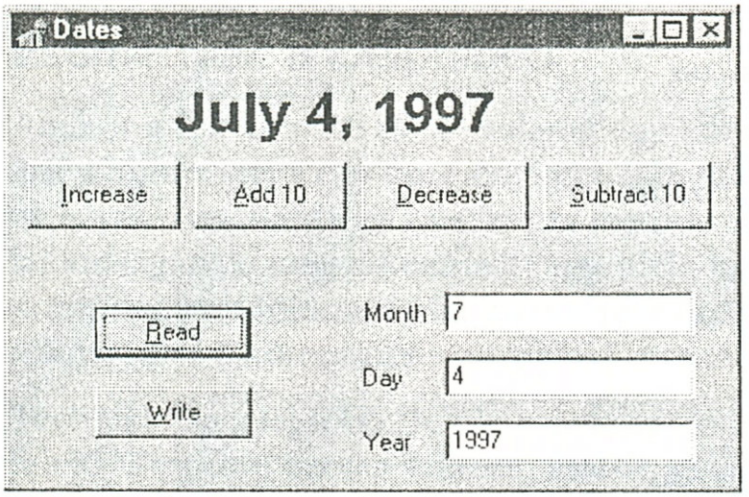
A tulajdonság definíciójában észrevehető, hogy néhányuk nem hivatkozik közvetlenül az adatra, egyszerűen kiértékelődik tartalmuk. Ez nem csak az olvasható tulajdonságok esetében van így, de az írható és olvasható tulajdonságoknál is. Gyakorlatilag kibővíthetnénk ezt a példát egy SetText metódus hozzáadásával, ami a stringet dátummá alakítja, és beállítja az FMonth, FDay, és FYear mezők új értékét.

Az osztály négy tulajdonsággal való kibővítése után átírhatjuk a példát, hogy amikor szükséges, használja is őket. Például használhatjuk a Text tulajdonságot:

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);
begin
  TheDay.Increase;
  LabelDate.Caption := TheDay.Text;
end;
```

A formhoz hozzáadtunk három szerkesztő dobozt, ahol a felhasználó írni és olvasni tudja a három fő tulajdonságot (ahogy azt a 6.7. ábrán is láthatjuk). Ez történik, ha megnyomunk egy gombot:

```
procedure TDateForm.ButtonReadClick(Sender: TObject);
begin
  EditYear.Text := IntToStr (TheDay.Year);
  EditMonth.Text := IntToStr (TheDay.Month);
  EditDay.Text := IntToStr (TheDay.Day);
end;
```



6.7. ábra *A DateProp példa frissített formja futási idő alatt*

Egy második gombnyomás az ellenkezőjét teszi:

```
procedure TDateForm.ButtonWriteClick(Sender: TObject);
begin
  TheDay.Year := StrToInt (EditYear.Text);
  TheDay.Month := StrToInt (EditMonth.Text);
  TheDay.Day := StrToInt (EditDay.Text);
  // update the label
  LabelDate.Caption := TheDay.Text;
end;
```

Ha kipróbáljuk ezt a példát (nem azt, amelyik a mellékelt CD-n található), akkor meglepő dolog történhet. Ugyanis, ha a jelenlegi objektum 5/1/1997-re hivatkozik, és megpróbálunk egy illegális dátumot beállítani, mint például 6/31/1998, akkor 6/1/1998-at kapunk! Azért van ez így, mert amikor beállítjuk az évet, a hónapot, és a napot, és kivételt váltunk ki, akkor átlépjük a rossz értéket, és az utána következő programsorokat, de az évet végül beállítjuk. Ez a megfelelő viselkedés, mivel mi jeleztük az osztály használóinak, hogy ellenőrizniük kell a dátum helyességét, mielőtt használnák. A dátum érvényességének ellenőrzése sok munkát igényel, mindazontáltal megpróbálhatjuk fejleszteni az osztályt.

A SetValue metódust írjuk tovább, hogy lekezelje ezt a problémát. Először eltároljuk a kezdeti értékeket, hogy ha kivétel keletkezik, akkor vissza tudjuk állítani az eredeti értékeket, a végén pedig újra egy kivételt váltunk ki, hogy a felhasználó lássa a hibaüzenetet:

```
procedure TDate.SetValue (m, d, y: Integer);
var
  OldY, OldM: Integer;
```

```
begin
  // store the old value
  OldY := FYear;
  OldM := FMonth;
  // passing the new value
  try
    FYear := y;
    // check the ranges
    SetMonth (m);
    SetDay (d);
  except
    on EDateOutOfRange do
    begin
      // reset the values
      FYear := OldY;
      FMonth := OldM;
      // let the error show up
      raise;
    end;
  end;
end;
```

A formon ezt a metódust használtuk a tulajdonságok közvetlen elérése helyett:

```
procedure TDateForm.ButtonWriteClick(Sender: TObject);
begin
  TheDay.SetValue (StrToInt (EditMonth.Text),
    StrToInt (EditDay.Text),
    StrToInt (EditYear.Text));
  // update the label
  LabelDate.Caption := TheDay.Text;
end;
```

Egyetlen dolog maradt már csak hátra. Az osztályoknak és a komponenseknek könnyen használhatónak kellene lenniük bonyolult előfeltételek nélkül. Sajnos, a TDate osztállyal nem ez a helyzet. Egy választható megoldás az lenne, hogy a három tulajdonságot csak olvashatóvá kellene tenni, így a programozók csak az általunk írt SetValue metódussal változtathatnák meg a dátumot. Ez nem egy rossz megoldás, de így nem jeleníthetjük meg a tulajdonságokat az Object Inspectorban (ahogy azt a fejezetben később tenni fogjuk). Egyelőre hagyjuk így a kódot, ahogy van, habár ez egy kissé kockázatos.

Megjegyzés

Az osztály SetValue metódusa és a három tulajdonsága közötti kapcsolat ugyan olyan, mint a TControl osztályok SetBounds metódusának, és a Left, Top, Width, és Height tulajdonságainak kapcsolata. Különleges körülmények között a fent leírt probléma merül fel a kontrolok pozícióját leíró tulajdonságoknál.

Események a Delphiben

Amikor a felhasználó csinál valamit egy komponenssel (például rákattint), akkor a komponens egy eseményt generál. Más eseményeket a rendszer hoz létre, olyanokat, mint egy választ egy metódushívásra, vagy a komponens egy tulajdonságának megváltozása (vagy akár egy másiké). Például, ha az egyik komponensre fókuszálunk, akkor az előzőleg aktuális komponens elveszti a fókusz, ami a komponens megfelelő eseményét futtatja le.

Technikailag a legtöbb Delphi esemény akkor következik be, amikor egy megfelelő Windows üzenet érkezik, ámbár az események nem egyeznek meg egy az egyben az üzenetekkel. Az események általában magasabb szintűek, mint a Windows üzenetek, ahogy például az egérrel való húzás esetében is (nincs meghatározott „egérrel húzás" nevű Windows üzenet).

Elméletileg egy esemény egy ablaknak küldött üzenet eredménye, és ez az ablak (vagy a megfelelő komponens) válaszolhat az üzenetre. Ezt a megközelítést folytatva, a TButton osztályt kellene alosztályosítanunk, hogy egy új eseménykezelőt adjunk hozzá, és hogy lekezeljünk a gombnyomás eseményt,

Gyakorlatban, egy új osztály létrehozása túlságosan bonyolult megoldás lenne. A Delphiben egy komponens eseménykezelője általában annak a formnak egy metódusa, amelyiken a komponens rajta van, és nem a sajátja. Más szavakkal, a komponens eseményeit az őt tartalmazó form kezeli. Ezt a technikát delegation-nek nevezik, és ez a Delphi komponens alapú modelljének alapja.

Az események tulajdonságok

Egy másik fontos fogalom, hogy az események tulajdonságok. Ez azt jelenti, hogy egy komponens eseményének kezeléséhez hozzá kell rendelnünk egy metódust a megfelelő esemény tulajdonsághoz, ahogy azt a fejezet CountOb2 példájában is tettük. Amikor duplán kattintunk egy eseményre az objektumvizsgálóban, egy újabb metódus adódik a tulajdonos formhoz, és az hozzárendelődik a komponens megfelelő eseménytulajdonságához.

Ez az oka annak, hogy több esemény is megoszthatja ugyanazt az eseménykezelőt, vagy futási idő alatt megváltoztathatja azt. Nincs szükségünk a nyelv alaposabb ismeretére ahhoz, hogy ezt kihasználjuk. Valójában, amikor kiválasztunk egy eseményt az Object Inspectorban, akkor az esemény nevének jobb oldalán található gombot megnyomva egy legördülő listában láthatjuk a „kompatíbilis" metódusokat – egy olyan listát, amelyben a metódusok ugyanolyan metódus-mutató típusúak. Az Object Inspectort használva könnyen kiválaszthatjuk ugyanazt a metódust más komponensek ugyanolyan eseményeihez, vagy egy komponens különböző, kompatíbilis eseményeihez.

Megjegyzés

Az Object Inspector az osztálydefiníció forráskódjából olvassa ki a kompatíbilis metódusok listáját. Ha egy új published metódust adunk a form osztály deklarációjában az inicializáló részhez (a public kulcsszó előtt) egy érvényes esemény paramétereivel, akkor az automatikusan megjelenik az Object Inspector listájában, a kiválasztott esemény kompatíbilis esemény kezelői között.

Egy esemény hozzáadása a TDate osztályhoz

Ahogy hozzáadtunk néhány tulajdonságot a TDate osztályhoz, úgy egy eseményt is hozzáadhatunk. Ez egy nagyon egyszerű esemény lesz: OnChangenek fogjuk hívni, és figyelmeztetni fogja a komponens használóját, hogy megváltozott a dátum. Egy esemény definiálásához egyszerűen egy megfelelő tulajdonságot, és néhány adatot adunk hozzá, hogy tárolni tudjuk az aktuális metódus-mutatót, amire az esemény hivatkozik. Ezek az új definíciók, amelyet az osztályhoz adtunk:

```
type
  TDate = class
  private
    FOnChange: TNotifyEvent;
    ...
  protected
    procedure DoChange; virtual;
    ...
  public
    property OnChange: TNotifyEvent
      read FOnChange write FOnChange;
    ...
  end;
```

A tulajdonság definiálása valójában nagyon egyszerű. Az osztály használója új értéket rendelhet a tulajdonsághoz, és ennél fogva az FOnChange privát mezőhöz. Az osztály nem rendel az FOnChange mezőhöz értéket, ezt az osztály használója teszi meg. A TDate osztály egyszerűen meghívja az FOnChange mezőben tárolt metódust, amikor a dátum értéke megváltozik. Természetesen ez csak akkor kerül meghívásra, ha az esemény tulajdonságot hozzárendeljük. Ezt a DoChange metódus hajtja végre:

```
procedure TDate.DoChange;
begin
  if Assigned (FOnChange) then
    FOnChange (Self);
end;
```

Ez a metódus mindig meghívódik, ha az értékek közül valamelyik megváltozik. Például:

```
procedure TDate.SetMonth (Value: Integer);
begin
  if (Value >= 1) and (Value <= 12) then
  begin
    FMonth := Value;
    DoChange;
  end
  else
    raise EDateOutOfRange.Create ('Month out of
        range');
end;
```

Megváltoztattuk az osztály mindkét metódusát, amelyek ugyanúgy változtatják a dátumot. A Year tulajdonsághoz hozzáadtunk egy write eljárást is, mivel az év megváltoztatásának is el kell indítania az eseményt:

```
procedure TDate.SetYear (Value: Integer);
begin
  FYear := Value;
  DoChange;
end;
```

Ha most megnézzük ezt az osztályt használó programot, akkor látszik, hogy jelentősen egyszerűsíthetjük azt. Először a form osztályhoz adunk hozzá egy saját metódust:

```
type
  TDateForm = class(TForm)
    ...
    procedure DateChange(Sender: TObject);
```

A metódus frissíti a címkét a TDate objektum Text tulajdonságának aktuális értékével:

```
procedure TDateForm.DateChange;
begin
  LabelDate.Caption := TheDay.Text;
end;
```

Utána ezt az eseménykezelőt beépítjük a FormCreate metódusba:

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TDate.Init (7, 4, 1997);
  LabelDate.Caption := TheDay.Text;
  // assign the event handler for future changes
  TheDay.OnChange := DateChange;
end;
```

Eléggé sok munkának tűnhet mindez. Hazudtam volna, amikor azt állítottam, hogy az eseménykezelő sok plusz munkától kímél meg minket? Nem. Most már egyáltalán nem kell foglalkoznunk a címke frissítésével, ha az objektum valamelyik adatát megváltoztatjuk. Itt az egyik gomb OnClick eseménye példaként:

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);
begin
  TheDay.Increase;
end;
```

Ugyanilyen egyszerűsített kód van sok más eseménykezelőben is. Ha egyszer már megírtuk az eseménykezelőt, akkor már nem kell folyamatosan frissítenünk a címkét, amely potenciális hibaforrás a programban. Eleinte sokat kell kódolnunk, mert ez nem egy Delphi komponens, hanem csak egy osztály. Egy komponensnél egyszerűbb a dolgunk, mert az Object Inspectorban csupán ki kell választani az eseménykezelőt, és be kell ínunk egy sort, hogy beállítsuk a címkét.

Mennyire nehéz egy új komponenst létrehozni Delphiben? Valójában egyszerű, ahogy ezt a következő részben látni fogjuk.

Megjegyzés

Ez a rész a tulajdonságok és események szerepének, és a komponensek írásának rövid bemutatása volt. Ezeknek az ismereteknek az alapvető ismerete minden Delphi programozónak fontos. Ha sokkal bonyolultabb komponenseket akarunk írni, akkor többet is megtudhatunk erről a témáról a 18. fejezetben.

Egy TDate komponens készítése

A következő lépésben a TDate osztályt komponenssé alakítjuk. Először az alapértelmezés szerinti TObject osztály helyett a TComponent osztályból kell örököltetnünk az osztályunkat.

```
type
  TDate = class (TComponent)
    ...
  public
    constructor Create (AOwner: TComponent); override;
```

Ahogy látjuk, egy új konstruktort is hozzá kellett adnunk az osztályhoz, fölülírva a komponensek eredeti konstruktorát, hogy megfelelő kezdőértéket biztosítsunk az objektumnak. Itt egyszerűen beállítjuk az aktuális dátum szerinti értékeket:

```
constructor TDate.Create (AOwner: TComponent);
var
  Y, D, M: Word;
begin
  inherited Create (AOwner);
  // today...
  DecodeDate (Now, Y, M, D);
  FYear := Y;
  FMonth := M;
  FDay := D;
end;
```

Ezek után az osztályunkat (a DATES.PAS fájl a DATECOMP könyvtárban) definiáló egységhez egy Register (regisztráló) eljárást kell hozzáadnunk. Erre azért van szükség, hogy a komponens a Delphi komponens palettájához hozzáadódjon. Az eljárást az egység interface részében kell deklarálnunk, nincs paramétere, és az implementációs részbe a következő sorokat írjuk:

```
procedure Register;
begin
  RegisterComponents ('Md3', [TDate]);
end;
```

Ez a kód egy új komponenst ad a paletta Md3-as lapjához, ha szükséges az oldalt is létrehozza. Egyébként ezt az oldalt használjuk az összes, a könyvben létrehozott komponensnek.

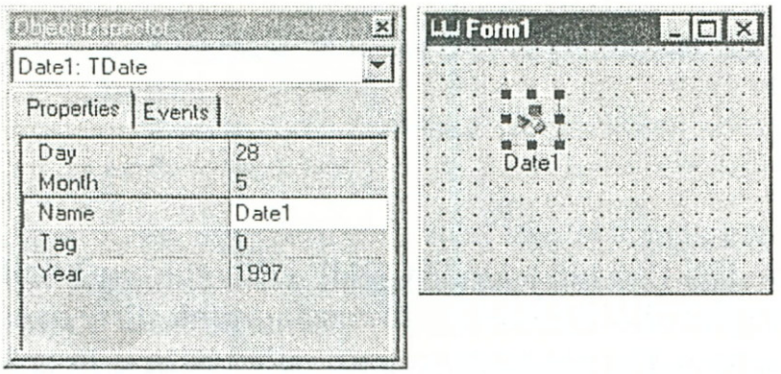
Az utolsó lépés a komponens installálása. Ennek az egyszerű komponensnek nem hozunk létre egy új csomagot. Ehelyett a Delphi alapértelmezés szerinti User's Components (felhasználók komponenseinek) csomagjába tesszük (a Delphi LIB könyvtárában található DCLUSR30.DPK fájl). Az új csomagok létrehozásáról a 18. fejezetben lesz szó.

Válasszuk ki a **Component** menü **Install Component** menüpontját, ott az **Into existing package** oldalt (ez az alapértelmezett), válasszuk ki a DCLUSR30.DPK csomag fájl nevet, és írjuk be a komponens egység fájl nevét: DATES.PAS. Az **OK** gomb megnyomása után a Delphi frissíti a csomagot, lefordítja, és megkér minket, hogy installáljuk fel (ha még nem tettük meg).

Ha most megnézzük a komponenspalettát, akkor ott lennie kell egy Md3 oldalnak, rajta az új komponenssel, ami a Delphi alapértelmezés szerinti új komponens ikonjával jelenik meg. Most már egy új applikáció formjára helyezhetjük, és a tulajdonságait módosíthatjuk az Object Inspectorban, ahogy azt a 6.8. ábrán is láthatjuk. Az OnChange eseményt sokkal könnyebben kezelhetjük, mint az előző példában.

Saját példaprogramokat is írhatunk ezt a komponenst használva (ami nagyon ajánlott), mindamellett most már megnézhetjük a DateComp példát, amely a

fejezetben lépésről lépésre felépített példánk újabb verziója. Alapjában véve ez egy egyszerűsített változata a DateEvt példának, mivel az eseménykezelő most már közvetlenül elérhető az Object Inspectorban.



6.8. ábra *Az új TDate komponensünk tulajdonságait láthatjuk az Object Inspectorban*

Hogyan tovább?

Ebben a fejezetben feltártuk az Object Pascal nyelv olyan komolyabb témáit, mint a metódus-mutatók, osztályhivatkozások, tulajdonságok és események, és a Delphi 3 új interfészei.

A következő lépésben a VCL hierarchia struktúráját vizsgáljuk meg. Ez zárja le a Delphi programozás alapjait, mielőtt nekivágnánk a különböző komponenseket használó példák elemzésének (amelyet a 8. fejezettől kezdünk). A következő fejezet a VCL struktúráját és a TObject osztály leírását tartalmazza. Valójában ez a speciális osztály a nyelv része, és nem a VCL-é, de a következő fejezetben tárgyaljuk, más alaposztályokkal együtt, mint például a TComponent, amit éppen most használtunk.

7. FEJEZET

A Vizuális Komponenskönyvtár

E fejezet tartalma:

- A TObject osztály
- A VCL hierarchiája
- Általános VCL tulajdonságok
- Általános VCL metódusok
- Általános VCL események
- A Delphi kollekcióinak használata
- A VCL forráskód tanulmányozása

A Vizuális Komponenskönyvtár (Visual Component Library)

A Delphi rendelkezik több, használatra kész standard rutinnal, de ezen kívül tartalmaz egy sokkal nagyobb és jelentősebb osztályhalmazt is. Az ebben található osztályok egy része komponens osztály, és megtalálható a komponenspalettán (Component Palette), míg más elemei általánosabb célúak. Ebben a fejezetben a központi témánk a Delphi osztálykönyvtára (class library) lesz, de megismerünk majd néhány általánosan használható elemet is. A komponensek használatáról, és más, a Windowshoz és az adatbázis-kezeléshez tartozó osztályokról a II. részben olvashatunk.

Annak ellenére, hogy nem csupán komponenseket tartalmaz, a Delphi rendszerkönyvtárát Visual Component Library-nek (Vizuális Komponenskönyvtárnak), vagy röviden VCL-nek nevezzük. Ez a fejezet a VCL rejtelmeibe nyújt bepillantást. Ha át szeretnénk lépni ezt a részt, akkor azt nyugodtan megtehetjük: lapozzunk át a következő fejezetre, amely egyszerűbb dolgokkal foglalkozik. Azonban ne felejtsünk el visszatérni, ha úgy érezzük, már teljesen tisztában vagyunk a Delphi alapjaival, és meg szeretnénk ismerni annak felépítését.

A TObject osztály

A Delphi működésének középpontjában egy osztályhierarchia áll. A rendszer minden osztálya a TObject osztály leszármazottja, így tehát a hierarchia minden tagjának egyetlen közös őse van. Ez azt teszi számunkra lehetővé, hogy a TObject adattípussal bármely, a Delphiben előforduló típust helyettesítsük. Nézzünk erre egy példát! Az eseménykezelőknek általában van egy TObject típusú Sender (küldő) nevű paramétere, ami annyit jelent, hogy a Sender objektum gyakorlatilag bármilyen osztályba tartozhat, hiszen minden osztály a TObject valamilyen szintű leszármazottja. Ennek a megközelítésnek viszont van egy hátránya is: ha dolgozni akarunk egy objektummal, akkor ismernünk kell annak konkrét adattípusát is, ellenkező esetben csak a TObject osztály metódusait és tulajdonságait használhatjuk a vele kapcsolatos műveleteknél. Ha ez a változó vagy paraméter például egy TButton típusú változót tartalmaz, akkor közvetlenül nem érhetjük el annak Caption tulajdonságát, hiszen ez nem a TObject osztály tulajdonsága. Ennek a problémának a megoldásához általában az `is` és az `as` RTTI (Run Time Type Information – futásidőben kiértékelt típusinformáció) operátorokat használjuk, amelyek segítségével meghatározhatjuk a kérdéses típusokat, és elvégezhetjük a szükséges típuskonverziókat. Visszatérve az előző példához, ha a TObject típusú Sender paraméter egy nyomógombra vonatkozik, akkor a következőket írhatjuk:

```
(Sender as TButton).Caption
```

Ennek a szerkezetnek megvan az a nagy előnye, hogy ha nem sikerül a konverzió, akkor a Delphi egy kivételt vált ki, amivel biztonságosabbá tehetjük programunkat.

Megjegyzés

Mivel az Object Pascal szintaxisában nincs konkrét eszköz sablon (általános, generic) osztályok (ahogy a C++ zsargon mondja) létrehozására, ezért ilyen célból a TObject típust, és az RTTI-t használjuk. Őszintén szólva nekem egyáltalán nem hiányoznak a C++ sablonok, az Object Pascal hozzáállása sokkal egyszerűbb, de ugyanolyan hatékony. Tudom, hogy a típus nélküli konténerek használata nem igazán biztonságos művelet a szükséges típuskonverziók miatt, de az RTTI és az ellenőrzött konverziók használatával minden probléma kivédhető. Az általános osztályokról még esik majd szó a fejezetben, a VCL konténerosztályairól szóló részben.

Mivel a TObject osztály „minden osztályok anyja", ezért talán nem árt egy pillantást vetnünk arra, hogy hogyan is van ez az eszköz a Delphi forráskódjában definiálva. A könnyebb érthetőség érdekében csoportosítottam a különböző eljárásokat, és megjegyzéseket is fűztem a sorokhoz:

```
type
  TObject = class
    // construction and destruction (commonly used)
    constructor Create;
    procedure Free;
    destructor Destroy; virtual;

    // construction and destruction (others)
    class function InitInstance(Instance: Pointer):
TObject;
    class function NewInstance: TObject; virtual;
    procedure CleanupInstance;
    procedure FreeInstance; virtual;

    // type information and parent classes
    class function InstanceSize: Longint;
    function ClassType: TClass;
    class function ClassInfo: Pointer;
    class function ClassName: ShortString;
    class function ClassNameLs
    class function ClassParent: TClass;
    class function InheritsFrom(AClass: TClass):
        Boolean;

    // message handling and dispatching
    procedure Dispatch(var Message);
    procedure DefaultHandler(var Message); virtual;
```

```
    // access to published elements
    class function MethodAddress(const Name:
        ShortString): Pointer;
    class function MethodName(Address: Pointer):
        ShortString;
    function FieldAddress(const Name: ShortString):
        Pointer;

    // interface support
    function GetInterface(const IID: TGUID; out Obj):
        Boolean;
    class function GetInterfaceEntry(const IID: TGUID):
        PInterfaceEntry;
    class function GetInterfaceTable: PInterfaceTable;
    function SafeCallException(ExceptObject: TObject;
      ExceptAddr: Pointer): Integer; virtual;
end;
```

A TObject osztály metódusai közül számosat bármely objektum használhat, még olyanok is, amiket magunk definiáltunk. Az eljárások nagy többségét azonban a rendszer használja, valamint akkor válnak lényegessé, amikor a Delphi fejlesztői környezetét kibővítő eszközöket készítünk.

Néhány TObject metódus akkor válik fontossá, amikor általános Windows alkalmazásokat készítünk. Vegyük például a ClassName metódust, ami egy string típusú változóban visszaadja az adott osztály nevét. Ez egy osztálymetódus, így mind osztályokra, mind pedig konkrét objektumpéldányokra is alkalmazhatjuk. Tegyük fel, hogy van egy TDate osztályunk, és egy abból származó Date1 objektumunk. A következő két kifejezés eredménye ebben az esetben megegyezik:

```
Text := Date1.ClassName;
Text := TDate.ClassName;
```

Elképzelhető, hogy egy osztály nevére vagyunk kíváncsiak, de fontos lehet egy osztály típusának, vagy szülőjének megállapítása is. Ezeket a műveleteket a ClassType és a ClassParent metódusokkal végezhetjük el. Ha már van egy osztályhivatkozásunk, attól kezdve azt úgy használhatjuk, mintha egy egyszerű objektum volna, például meghívhatjuk rá a ClassName metódust.

Egy másik fontos eljárás az InstanceSize, amely visszaadja egy objektum futásidő alatti méretét. Arra is gondolhatnánk esetleg, hogy itt a SizeOf globális függvényt használjuk, ez azonban nem az objektum, hanem az arra mutató pointer méretét adja vissza, ami mindig négy byte.

Az osztályinformáció megjelenítése

Készítettem egy kis példaprogramot a TObject osztály és metódusainak bemutatására. Ha magunk is el akarjuk készíteni ezt a programot, akkor a következőket kell tennünk: helyezzünk egy gombot és egy listadobozt(list box) egy új, üres alkalmazás formjára. Ha ezzel megvagyunk, adjuk hozzá a projekthez azt az egységet, ami az 5. fejezetben elkészített TDate osztályt definiálja, majd írjunk egy, az előbbi osztályra vonatkozó uses kifejezést az új projekt forráskódjába.

Most készítsünk egy új eljárást, ami egy általános objektumot kap paraméterül, és a Listbox komponensnek küldi el az adott objektum leírását. Valahogy így kell ennek a ShowInfo metódusnak kinéznie:

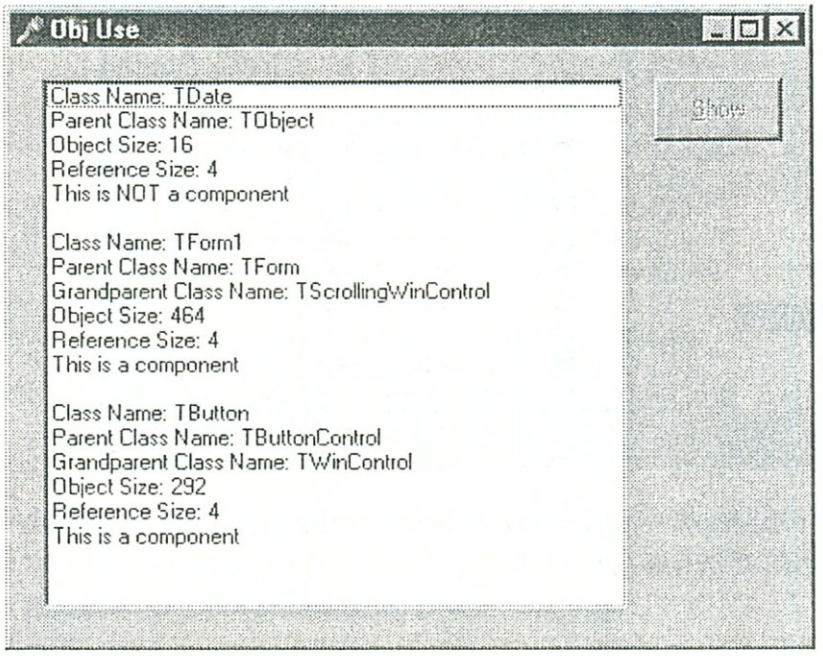
```
procedure TForm1.ShowInfo (Obj: TObject);
begin
  // add class name
  ListBox1.Items.Add ('Class Name: ' + Obj.ClassName);
  // add parent class name, if any
  if Obj.ClassParent <> nil then
  begin
    ListBox1.Items.Add ('Parent Class Name: ' +
      Obj.ClassParent.ClassName);
    // add grandparent class name, if any
    if Obj.ClassParent.ClassParent <> nil then
      ListBox1.Items.Add ('Grandparent Class Name: ' +
        Obj.ClassParent.ClassParent.ClassName);
  end;
  // add the size of object and reference
  ListBox1.Items.Add ('Object Size: ' +
    IntToStr (Obj.InstanceSize));
  ListBox1.Items.Add ('Reference Size: ' +
    IntToStr (SizeOf (Obj)));
  // indicate if this is a component or not
  if Obj.InheritsFrom (TComponent) then
    ListBox1.Items.Add ('This is a component')
  else
    ListBox1.Items.Add ('This is NOT a component');
end;
```

Most, ha a felhasználó a gombra kattint, meghívhatjuk a ShowInfo eljárást három különböző paraméterrel, a TDate, a Self (ez maga a form) és a Sender (ez a gomb, ami az esemény generálja) objektumokkal.

```
procedure TForm1.ShowButtonClick(Sender: TObject);
var
  Day: TDate;
begin
```

```
    {create an instance and show some information}
    Day := TDate.Init (6, 1, 1995);
    ShowInfo (Day);
    ListBox1.Items.Add ('');
    {show the same information about the form and the
        sender}
    ShowInfo (Self);
    ListBox1.Items.Add ('');
    ShowInfo (Sender);
    {free memory}
    Day.Free;
    {disable the button, to avoid a second click}
    ShowButton.Enabled := False;
end;
```

Ha lefuttatjuk a programot, a megjelenő ablakban elolvashatjuk a három osztály leírását, ahogy azt a 7.1. ábra is mutatja.



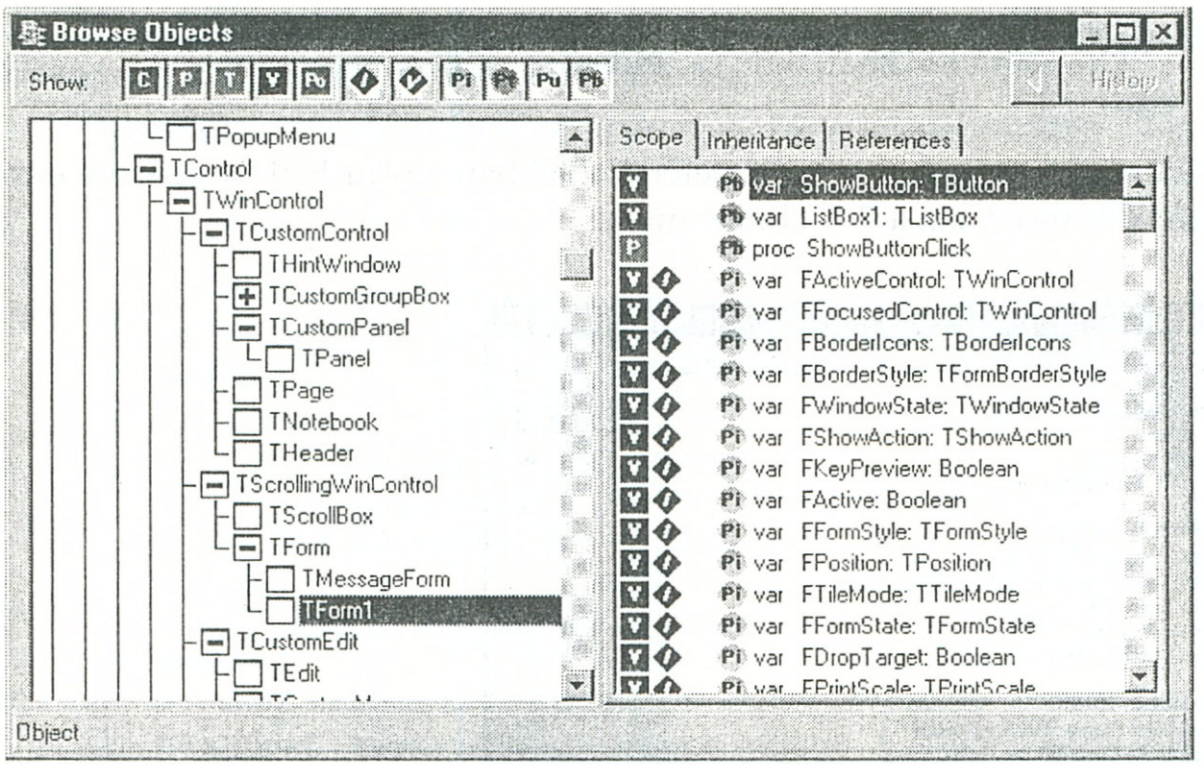
7.1. ábra *Az ObjUse példa végeredménye*

A VCL hierarchiája

A VCL számos, a TObject-ből származó alosztályt definiál. Ezeknek az osztályoknak nagy része más osztályok leszármazottja, így ez a rendszer egy igen összetett hierarchiát alkot. Attól az esettől eltekintve, mikor egy új komponens kifejlesztésén dolgozunk, mi csupán a hierarchia végosztályaival – az osztályok felépítését ábrázoló fa leveleivel – fogunk dolgozni. Ezeknek a

komponenseknek a teljes dokumentációja megtalálható a Delphi Súgójában. Ha esetleg a rendszer teljes szerkezetére vagyunk kíváncsiak, akkor használjuk az Object Browsert (ezt az eszközt egyébként részletesen bemutatjuk a 26. fejezetben), nézzünk utána a Borland és más cégek kiadványaiban, vagy tanulmányozzuk a fejezetben található ábrákat. A 7.2. ábra az Object Browsert mutatja működés közben.

A VCL hierarchiát három fő területre oszthatjuk fel: a komponensekre, az általános objektumokra, és a kivételekre.



7.2. ábra *Így néz ki az Object Browser (Objektumtallózó), amint éppen a VCL hierarchiát mutatja*

Megjegyzés

A VCL osztályok teljes felépítését tanulmányozhatjuk, ha lefuttatjuk a VclHiera példát a Tools könyvtárból. További ide vonatkozó információk a CD Readme fájljában találhatók.

A komponensek

A komponensek a Delphi alkalmazások központi összetevői. Delphiben a programírás legtöbbször nagyjából annyit jelent, hogy komponensek között válogatunk, és meghatározzuk azok tulajdonságait.

A Delphi komponenseknek több válfaja is létezik. Legtöbbjük megtalálható a komponenspalettán, de nem mind, olyanok, mint például a TForm, vagy a TApplication nem. Gyakorlatilag a komponensek a TComponent osztály leszármazottjai. Mint ilyenek, átirányíthatók DFM fájlokba (ez a tulajdonság abból következik, hogy a komponensek a TPersistent osztályból örökölnek), és vizuális módszerekkel állítható eseményeik és tulajdonságaik vannak. Az előző fejezet végén láthattunk arra egy egyszerű példát, hogy hogyan kell komponenseket készíteni.

Megjegyzés

A Delphi komponensek nagy részét példákban is fogjuk használni a könyvben. Megpróbáltam logikai csoportokba rendezni őket, minden fejezet egy ilyen csoportot ír le, ám ez a csoportosítás csak részben egyezik meg a komponenspaletta oldalaiéval.

A 7.3. ábra a VCL felépítésének egy részét mutatja be. A rajz nem teljes, hiszen a hierarchia bonyolultabb annál, minthogy egyetlen oldalra ráférjen, és a célom is inkább a rendszer általános struktúrájának bemutatása.

Ahogy a 7.4. ábrán is láthatjuk, a komponensek különálló csoportokba rendezhetők, és neveik tükrözik azok belső tartalmát. Lássuk most, hogy milyen csoportokba rendezhetjük őket:

- A kontrolok (control), vagy más néven vizuális komponensek (visual component) mind a TControl osztály leszármazottjai. A kontroloknak adott méretük és helyzetük van a képernyőn, és tervezési időben ugyanott helyezkednek el, ahol majd futási időben fognak. Két fajtájuk létezik, az ablak alapúak, és a grafikusak:
  - Az ablak alapú (window-based, vagy windowed) kontrolok olyan vizuális komponensek, amik rendszerablakokon alapulnak. Technikai szempontból ez azt jelenti, hogy ezeknek az összetevőnek van ablak leírójuk (window handle), felhasználói szempontból pedig azt, hogy ezek a komponensek input fókuszba kerülhetnek, és tartalmazhatnak más kontrolokat. Ez a Delphi komponenscsoportjai közül a legnagyobb.

  - A grafikus (graphical), vagy nem ablak alapú (non-window based) kontrolok, ahogy azt a nevük is mutatja, olyan komponensek, amik nem ablakokon alapulnak. Ez azt jelenti, hogy nincs leírójuk, nem kaphatják meg a fókuszt, és nem tartalmazhatnak más kontrolokat. Ezeket az elemeket a szülő formjuk rajzolja ki, és tőle kapják az egér által generált, és más eseményeket. A Címke (Label) és a SpeedButton komponensek jó példák a nem ablak alapú kontrolokra. Kevés kontrol tartozik ebbe a csoportba, de azok nagyon fontosak abból a szempontból, hogy segítenek a számítógép erőforrásainak takarékosabb kihasználásában, különösen akkor, ha nagy számban alkalmazzuk őket.

- A nem vizuális (nonvisual) komponensek közé mindazok az elemek tartoznak, amik nem kontrolok; mindazok az összetevők, amelyek a TComponent, de nem a TControl osztály leszármazottjai. Tervezéskor a nem vizuális elemek ikonként (néha a feliratukkal együtt) jelennek meg a formon, futás közben egyesek láthatók (ilyen a standard dialógusablak), egyesek láthatatlanok (mint az adatbázis-táblázat (database table) komponens). Más szóval maguk a nem vizuális összetevők nem látszanak a program futása közben, de eredményezhetnek látható dolgokat, mint például a párbeszédablak.

Tipp

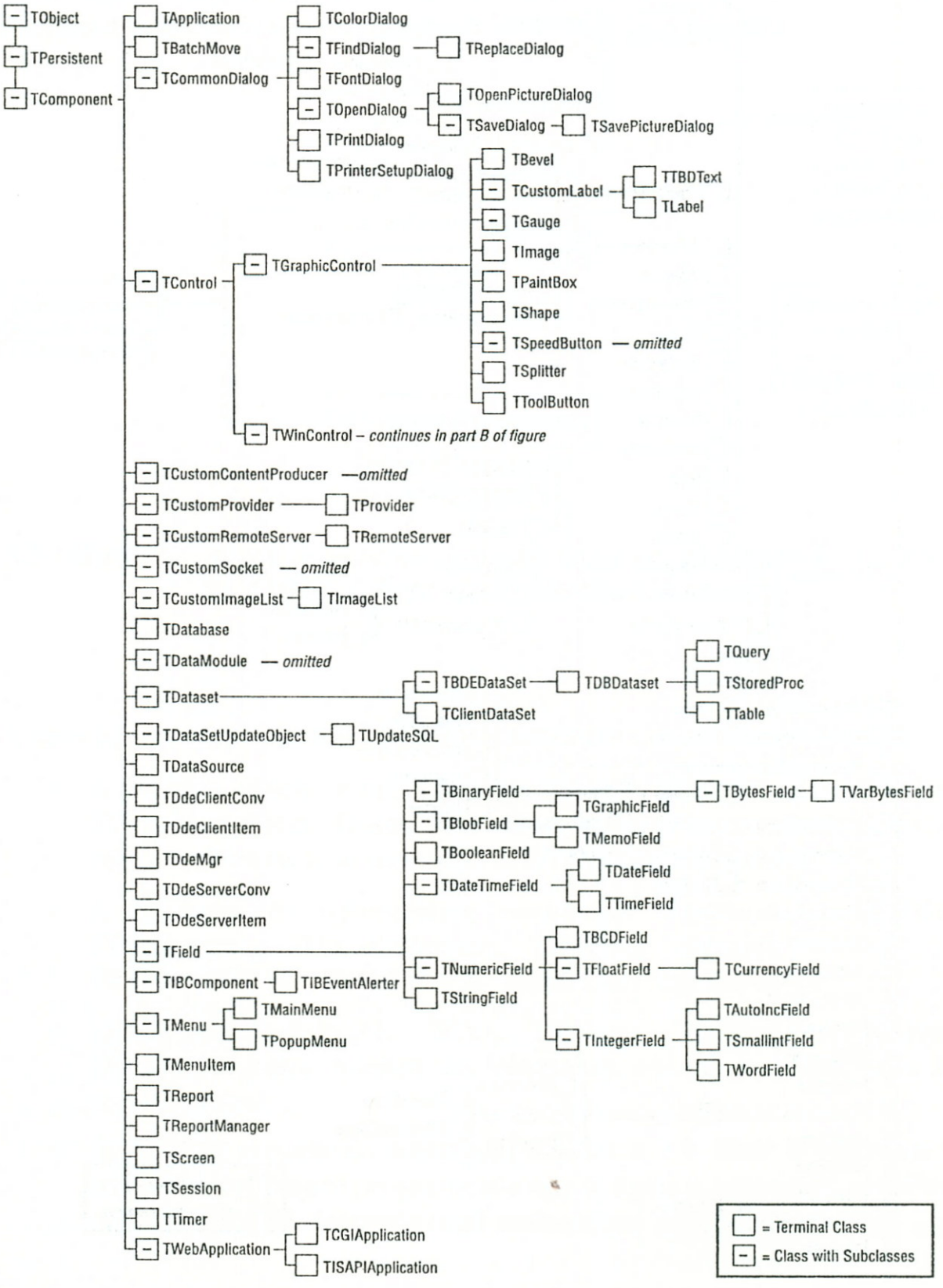
Használhatunk egy környezeti beállítást, a **Show Component Captions** pontot, ha látni szeretnénk a komponensek nevét ikonjaik alatt. Így könnyebb lesz őket megtalálni, főleg, ha már sok egyforma komponensünk van a formon.

Az objektumok

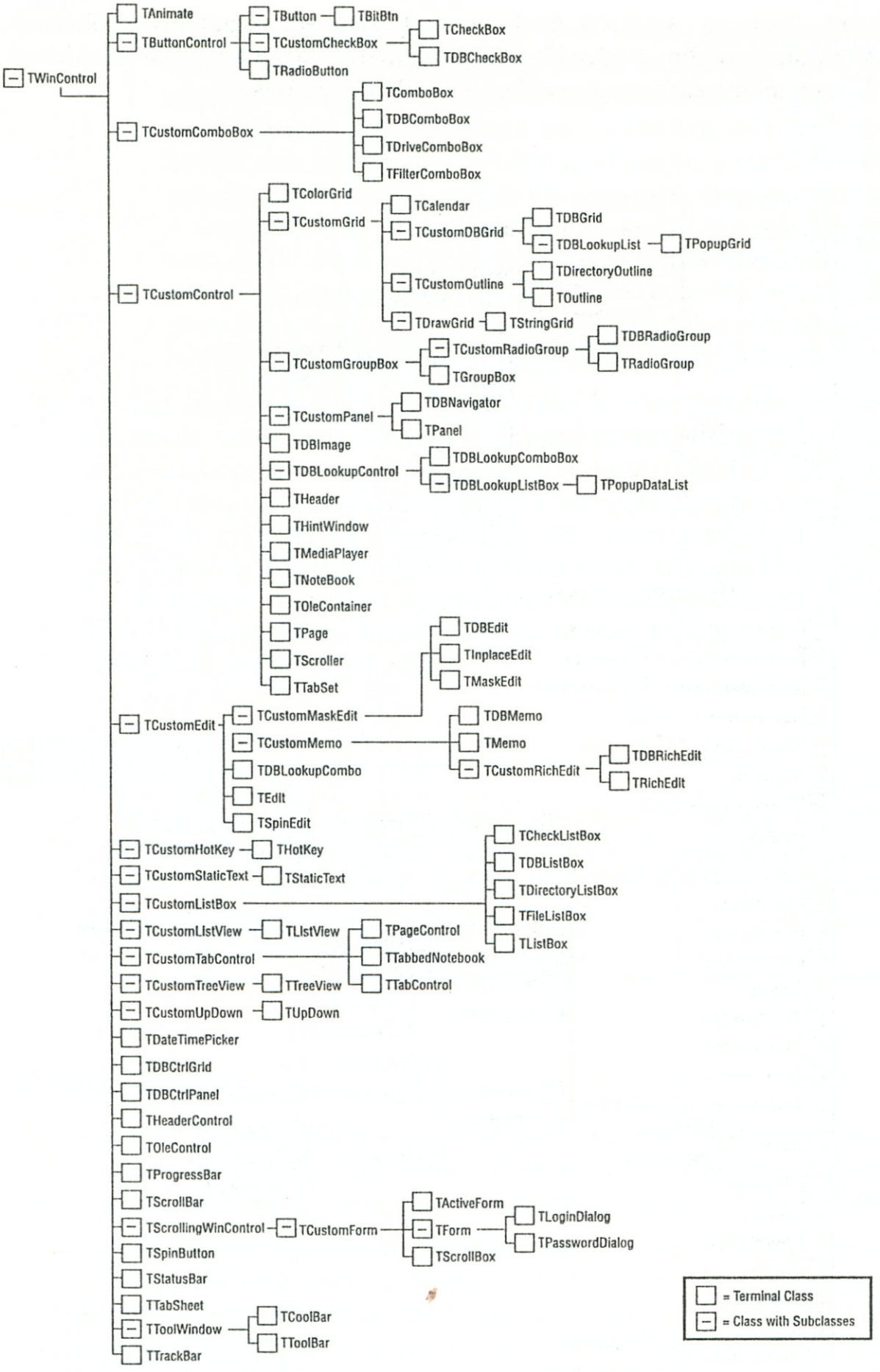
Ugyan a VCL alapvetően komponensek gyűjteménye, tartalmaz olyan osztályokat is, amik nem férnek bele ebbe a kategóriába, mert nem a TComponent osztály leszármazottjai. Ezen osztályok legfontosabbjait a 7.5. ábrán láthatjuk.

Bár a Delphi Súgója és más források objektumként (object) hivatkoznak a nem a TComponentből származó osztályokra, ez az elnevezés nem egészen helyénvaló. Ezeknek az osztályoknak két fő felhasználási területük van. Az első az, hogy a komponensek tulajdonságainak típusokat definiálnak, olyanokat, mint a Picture tulajdonság egy kép (image) komponensnek (ami egy TGraphic objektum), vagy az Item tulajdonság egy list boxnak (ami pedig egy TStrings

objektum). Ezek az osztályok általában a TPersistent osztályból örökölnek, aminek eredményeként átirányíthatók (streamable) lesznek, és al-tulajdonságokkal, vagy akár saját eseményekkel is rendelkezhetnek.

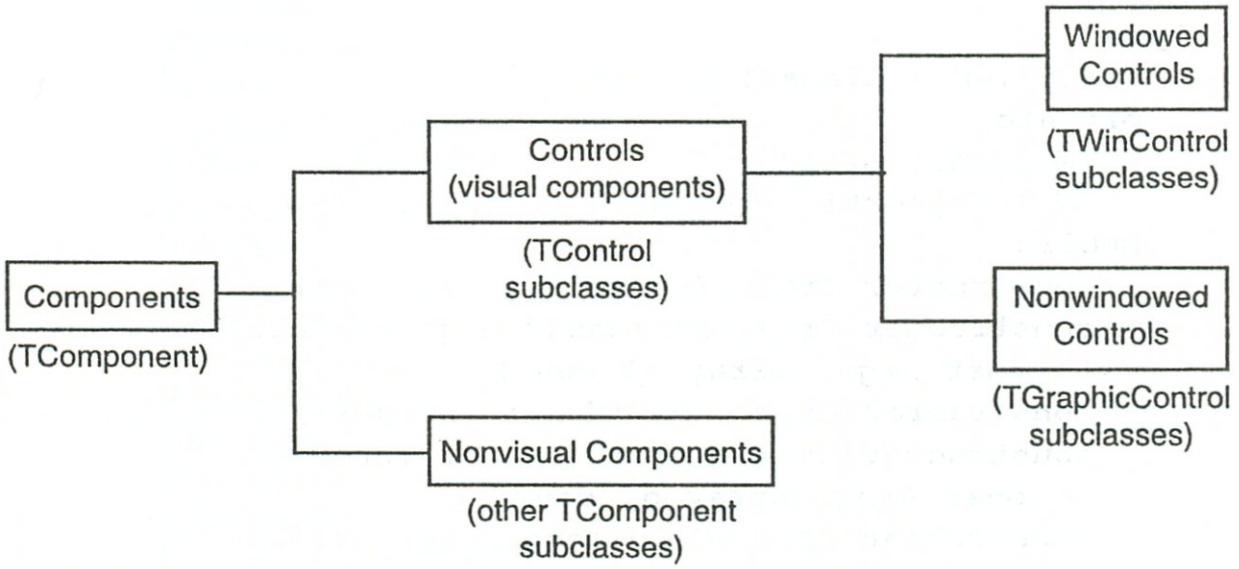


7.3. ábra *A VCL hierarchia első része: a komponensek*



7.4. ábra *Íme a komponensek csoportjának rajzos reprezentációja*

A nem-komponens osztályok második felhasználási területe a közvetlen használat. A Delphi programokban szabadon lefoglalhatunk és használhatunk ilyen típusú osztályokat. Ezt több céllal is megtehetjük: például egy objektum tulajdonságait tárolhatjuk, és megváltoztathatjuk egy ilyen elemben anélkül, hogy annak eredeti értékeihez nyúlnánk, vagy akár értékek egész listáját elraktározhatjuk komplexebb algoritmusok írásához. A könyv sok példát fog majd mutatni a nem-komponens osztályok ilyen célú használatára.



7.5. ábra *A VCL hierarchia második része: az objektumok. A B rész az új, COM-hoz kapcsolódó osztályokat mutatja*

A nem-komponens osztályokat több különböző csoportba sorolhatjuk:

- A grafikus objektumok: TBitmap, TBrush, TCanvas, TFont, TGraphic, TGraphicsObject, TIcon, TMetafile, TPen, TPicture. Ezek közül néhányat részletesen is ismertetünk a 10. és a 11. fejezetben.
- A csatorna- és fájlkezelésre vonatkozó objektumok: TBlobStream, TFileStream, THandleStream, TIniFile, TMemoryStream, TFiler, TReader, TWriter. Ezekkel a 29. fejezetben foglalkozunk.
- Listák és kollekciók: TList, TStrings, TStringList, TCollection, TCollectionItem. Ezeket az elemeket ebben a fejezetben fogjuk megtárgyalni.
- A COM-ra vonatkozó osztályok, amiket a 7.5. ábra B részében láthatunk. Ezek a Delphi programozás egy új ágához tartoznak, ezzel a témával a 21-től a 24. fejezetig tartó szakaszban foglalkozunk majd.

A kivételek

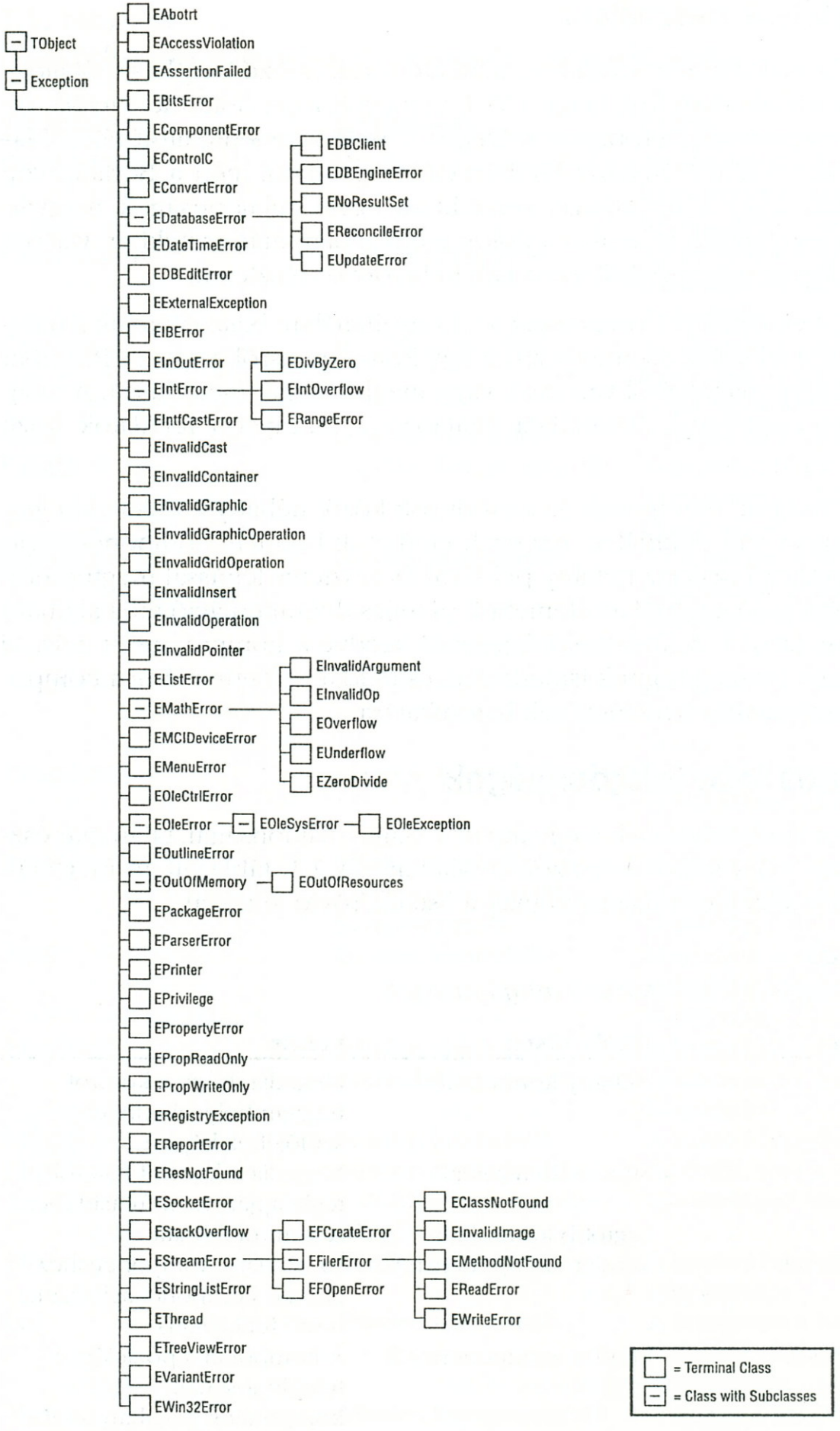
A VCL osztályok harmadik nagy csoportját a kivételek (exceptions) alkotják. A kivételkezelést már megbeszéltük az 5. fejezetben, így most itt nem kívánok részletesebb leírásba bocsátkozni. A kivételosztályok felépítését a 7.6. ábra mutatja.

Minden kivétel az Exception osztály leszármazottja. Az osztály definíciója a következő:

```
type
  Exception = class(TObject)
  private
    FMessage: string;
    FHelpContext: Integer;
  public
    constructor Create(const Msg: string);
    constructor CreateFmt(const Msg: string;
      const Args: array of const);
    constructor CreateRes(Ident: Integer);
    constructor CreateResFmt(Ident: Integer;
      const Args: array of const);
    constructor CreateHelp(const Msg: string;
      AHelpContext: Integer);
    constructor CreateFmtHelp(const Msg: string;
      const Args: array of const; AHelpContext:
      Integer);
    constructor CreateResHelp(Ident: Integer;
      AHelpContext: Integer);
    constructor CreateResFmtHelp(Ident: Integer;
      const Args: array of const; AHelpContext:
      Integer);
    property HelpContext: Integer
      read FHelpContext write FHelpContext;
    property Message: string
      read FMessage write FMessage;
  end;
```

Két dolgot kell megemlítenem a definícióval kapcsolatban. Az első az, hogy a sorok tele vannak különböző konstruktorokkal, melyek stringekkel, a stringek forrásazonosítójával, formátumstringekkel, a forrásokból származó formátumstringekkel, és ezek kombinációival vannak paraméterezve. A második fontos dolog a Message (üzenet) tulajdonság, amellyel a konstruktorokban definiált üzeneteket lehet elérni. Egyes konstruktorok még a HelpContext tulajdonságot is beállítják.

Ha látni szeretnénk, hogyan működnek ezek az elemek, akkor lapozzunk viszsza az 5. fejezet végén található példához. A könyv többi olyan részein is találkozhatunk kivétel-kezelési példákkal, ahol az erőforrások lefoglalásáról esik szó.



7.6. ábra *A VCL hierarchia harmadik része: a kivételek*

A VCL osztályok használata

A VCL osztályok dokumentációja megtalálható a különböző Delphi súgófájlokban. Először azt akartam írni, hogy a VCL osztályok *teljes* dokumentációja, de ebben nem vagyok teljesen biztos: a Delphi 1 és 2 sajnos mutatott némi hiányosságot ezen a téren. Néhány esetben ez elfogadható, mert a Borland cég úgy gondolta, hogy a következő verziókban egyes tulajdonságok megváltoznak, így nem fixálják le azokat a dokumentációban, más esetekben viszont egyszerűen figyelmetlenségből maradtak ki bizonyos részletek.

Minden VCL-beli osztályra vonatkozik az, hogy általában használhatjuk a tulajdonságait, lekezelhetjük eseményeit (ha egy komponensről, vagy a TPersitent osztály egy leszármazottjáról van szó), vagy meghívhatjuk metódusait. A tulajdonságok és események használata általában a könnyebb feladatok közé tartozik.

Ezen tulajdonságok, események és metódusok közül néhány a hierarchia legmagasabb fokán van definiálva, így ezek elérhetők bármely komponens számára. A következő szakasz néhány példával és részletes leírással megfűszerezett áttekintést nyújt ezekről az elemekről (a teljes dokumentáció megtalálható a Delphi Súgójában). A következő fejezettől kezdve a komponensek más, új tulajdonságaival is meg fogunk ismerkedni, és példákat fogunk látni a komponenspaletta szinte minden elemének használatára.

Az általános VCL tulajdonságok

Noha minden komponensnek megvannak a maga tulajdonságai, talán már észrevettük, hogy ezek között sok azonos található. A 7.1. táblázatban megtalálhatjuk néhány általános tulajdonságnak a listáját, rövid leírással.

7.1. táblázat

A legtöbb komponensnél elérhető tulajdonságok

Tulajdonság	Elérhetőség	Leírás
Align	Néhány kontrolnál	Megadja, hogy a kontrol hogyan helyezkedik el a szülőjéhez képest
BoundsRect	Minden kontrolnál	Megadja a kontrol határoló téglalapját (csak futásidőben)
Caption	Legtöbb kontrolnál	A kontrol felirata
ComponentCount	Minden komponensnél	Az aktuális komponenshez tartozó komponensek száma (csak futásidejű)
ComponentIndex	Minden komponensnél	A komponens pozíciója a tulajdonosának komponenslistájában (csak futásidejű)

7.1. táblázat

A legtöbb komponensnél elérhető tulajdonságok (folytatás)

Tulajdonság	Elérhetőség	Leírás
Components	Minden komponensnél	Az aktuális komponens komponenseit tartalmazó tömb (csak futásidejű)
ControlCount	Minden kontrolnál	Az aktuális kontrol gyermek-kontroljainak száma (csak futásidejű)
Controls	Minden kontrolnál	Az aktuális kontrol gyermek-kontroljait tartalmazó tömb (csak futásidejű)
Color	Legtöbb kontrolnál	A háttér színét adja vissza
Ctrl3D	Legtöbb komponensnél	Visszaadja, hogy a komponens külalakja három dimenziós-e
Cursor	Minden kontrolnál	Az a kurzor, amit akkor használunk, amikor az egérmutató a kontrol fölött van
DragCursor	Legtöbb kontrolnál	Az a kurzor, ami azt jelzi, hogy a kontrolt arrébb lehet húzni
DragMode	Legtöbb kontrolnál	A drag and drop stílusú viselkedését definiálja a kontrolnak
Enabled	Minden kontrolnál, és néhány nem vizuális komponensnél	Megadja, hogy a kontrol aktív, inaktív, vagy elszürkített
Font	Minden kontrolnál	Megadja a komponens belsejében megjelenő szöveg betűtípusát
Handle	Minden ablak alapú kontrolnál	A kontrol által használt rendszerablak leírója (csak futásidejű)
Height	Minden kontrolnál	A kontrol függőleges mérete
HelpContext	Minden kontrolnál és a dialógus komponen-seknél	Az érték, amit a szövegkör-nyezet-függő Súgó használ
Hint	Minden kontrolnál	A kontrol felett megjelenő tipp szövege
Left	Minden kontrolnál	A komponens bal felső sarkának vízszintes koordinátája
Name	Minden komponensnél	A komponens példányának egyedi neve, amit általában a forráskód használ

7.1. táblázat

A legtöbb komponensnél elérhető tulajdonságok (folytatás)

Tulajdonság	Elérhetőség	Leírás
Owner	Minden komponensnél	A komponens tulajdonosát adja vissza (csak futásidejű)
Parent	Minden kontrolnál	A szülő kontrolt adja vissza (csak futásidejű)
ParentColor	Legtöbb kontrolnál	Megadja, hogy a kontrol a szülőjével megegyező színt használ-e
ParentCtl3D	Legtöbb komponensnél	Megadja, hogy a komponens a szülőjével megegyező Ctl3D-t használ-e
ParentFont	Minden kontrolnál	Megadja, hogy a komponens a szülőjével megegyező betűtípust használ-e
ParentShowHint	Minden kontrolnál	Megadja, hogy a komponens a szülőjével megegyező ShowHint beállítást használ-e
PopupMenu	Minden kontrolnál	Az előugró menü, amit meg kell jeleníteni, ha a kontrolon az egér jobb gombjával kattintanak
ShowHint	Minden kontrolnál	Megadja, hogy a tippek megjelenítése engedélyezve van-e
Showing	Minden kontrolnál	Megadja, hogy a kontrol éppen látható-e a képernyőn, és hogy a szülője, a szülőjének szülője, stb. is láthatók-e
TabOrder	Minden ablak alapú kontrolnál	A szülő kontrol által meghatározott tabulátor-sorrendet adja vissza
TabStop	Minden ablak alapú kontrolnál	Megadja, hogy a felhasználó elhagyhatja-e a kontrolt a TAB billentyűvel
Tag	Minden komponensnél	Egy longint, amelyet tetszőleges érték tárolására használhatunk
Top	Minden kontrolnál	A komponens bal felső sarkának vízszintes koordinátája
Visible	Minden kontrolnál	Megadja, hogy a kontrol látható-e (lásd még a Showing tulajdonságot)
Width	Minden kontrolnál	A kontrol vízszintes mérete

Mivel a komponensek között öröklődési kapcsolat áll fönn, érdekes lehet utánanézni, hogy vajon melyik osztályból származik a tulajdonságok nagy része. A 7.7. ábrán láthatjuk a VCL hierarchia által bevezetett tulajdonságok összefoglalását. A következő részek a legalapvetőbb tulajdonságok részletes leírását tartalmazzák.

A Name (Név) tulajdonság

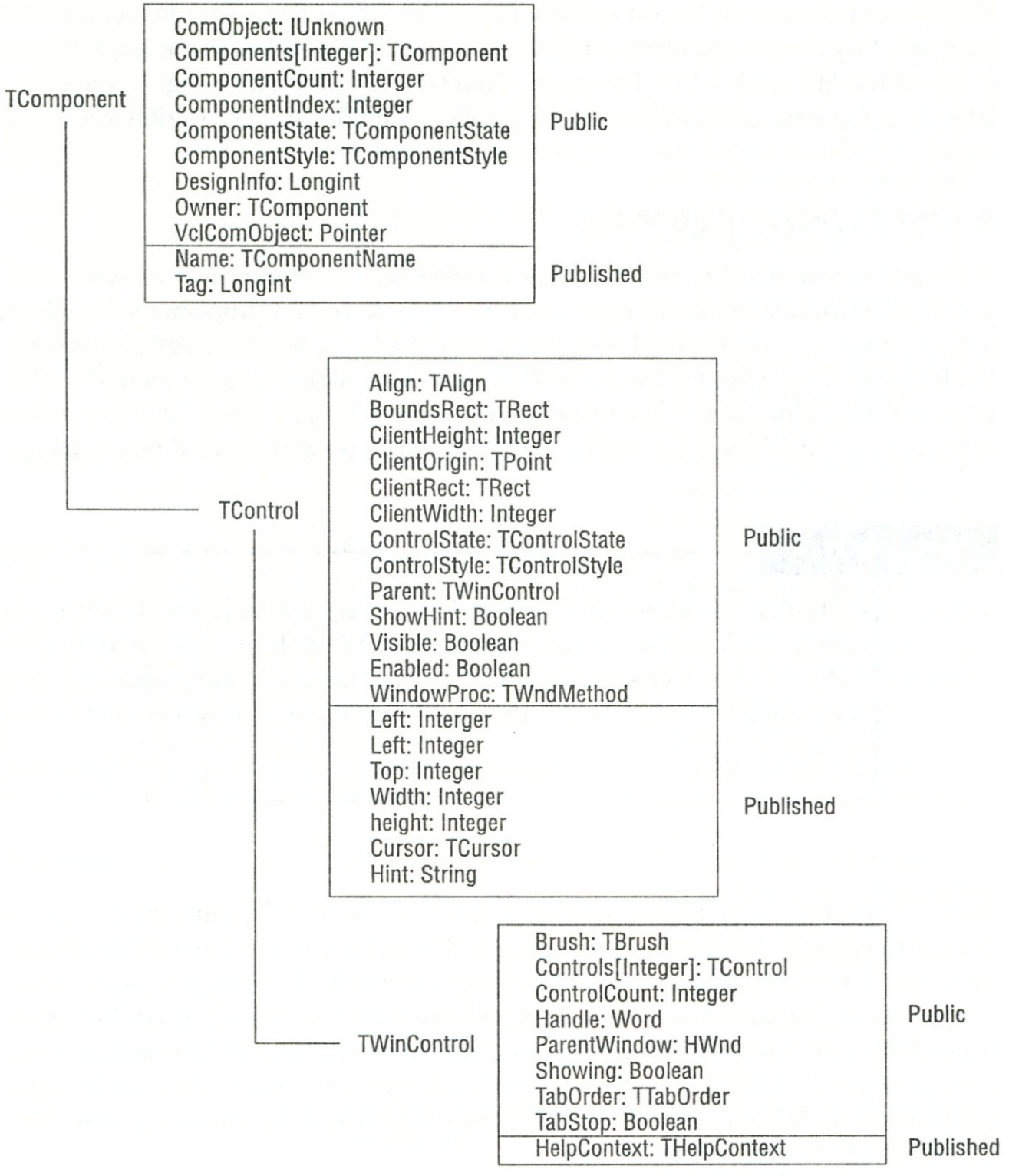
A Delphiben minden komponensnek rendelkeznie kell egy névvel. A névnek a szülő komponensen belül egyedinek kell lennie (ez a komponens általában a form, amire az összetevőt helyezzük). Ez azt jelenti, hogy ha egy alkalmazás két formján van egy-egy azonos nevű komponens, akkor az nem hiba, de ajánlatos ezt elkerülni, hiszen így könnyen összekeverhetjük őket. Általában tehát ajánlatos egy alkalmazáson belül minden komponensnek egyedi nevet adni.

Megjegyzés

Technikailag lehetséges név nélküli komponensek létrehozása, de ennek csak dinamikusan létrehozott elemeknél van értelme. A Delphi a komponensek nevével azonosítja azok megfelelő mezőit, ezért soha ne változtassuk meg egyetlen összetevő nevét sem futásidő alatt!

Nagyon fontos, hogy helyes értéket adjunk a Name tulajdonságnak. Ha túl hosszúra választjuk a nevet, akkor sokat kell majd gépelnünk programírás közben, ha túl rövidre, akkor nehéz lesz megjegyeznünk, és könnyen összekeverhetjük más nevekkel. Általában a komponens neve a típusának kezdőbetűjével kezdődik; ez a konvenció segít abban, hogy könnyebben olvashassuk a programszöveget, és lehetővé teszi, hogy a Delphi típusai szerint rendezze komponenseinket az Object Inspectorban. Három fontos szabály van még, amit névadáskor figyelembe kell vennünk:

- Mindenekelőtt, a Name tulajdonságot arra használjuk, hogy a form deklarációs részében definiáljuk a kívánt objektumot. Ezen a néven fogunk rá hivatkozni programíráskor, és így a névnek egy érvényes Pascal azonosítónak kell lennie, ahogy azt az 1. fejezet CheckId példájában láttuk.
- Másodszor, ha a Caption tulajdonság megváltoztatása előtt változtatunk a Name-en, akkor az a feliratra is hatással lesz, ahogy azt már az 1. fejezetben láttuk. Vagyis ha a név és a Caption megegyeznek, akkor a névváltoztatáskor a másik tulajdonság is megváltozik.



7.7. ábra *A tulajdonságok, amik a VCL hierarchia legfelsőbb szintjén kerülnek bevezetésre, elérhetők minden alosztály számára*

- Harmadszor, a Delphi a komponens nevéből fogja származtatni az ahhoz kapcsolódó események nevét. Ha van például egy Button1 gombunk, akkor a megnyomásából származó esemény kezelőjének neve Button1Click lesz, hacsak konkrétan meg nem adunk neki egy másik

nevet. Ha később megváltoztatjuk a komponens nevét, akkor a hozzá tartozó eseménykezelő neve is automatikusan megváltozik. Ha az előbbi példában a gomb nevét MyButton-ra változtatjuk, akkor az eljárás neve is Button1Click-ről MyButtonClick-re változik.

Mindenesetre ne feledjük, hogy amellett, hogy név szerint hivatkozunk egy komponensre, használhatjuk a hozzá tartozó form Components és Controls tulajdonságát is. Itt egy példaprogram, amiben ezek használatát mutatjuk be:

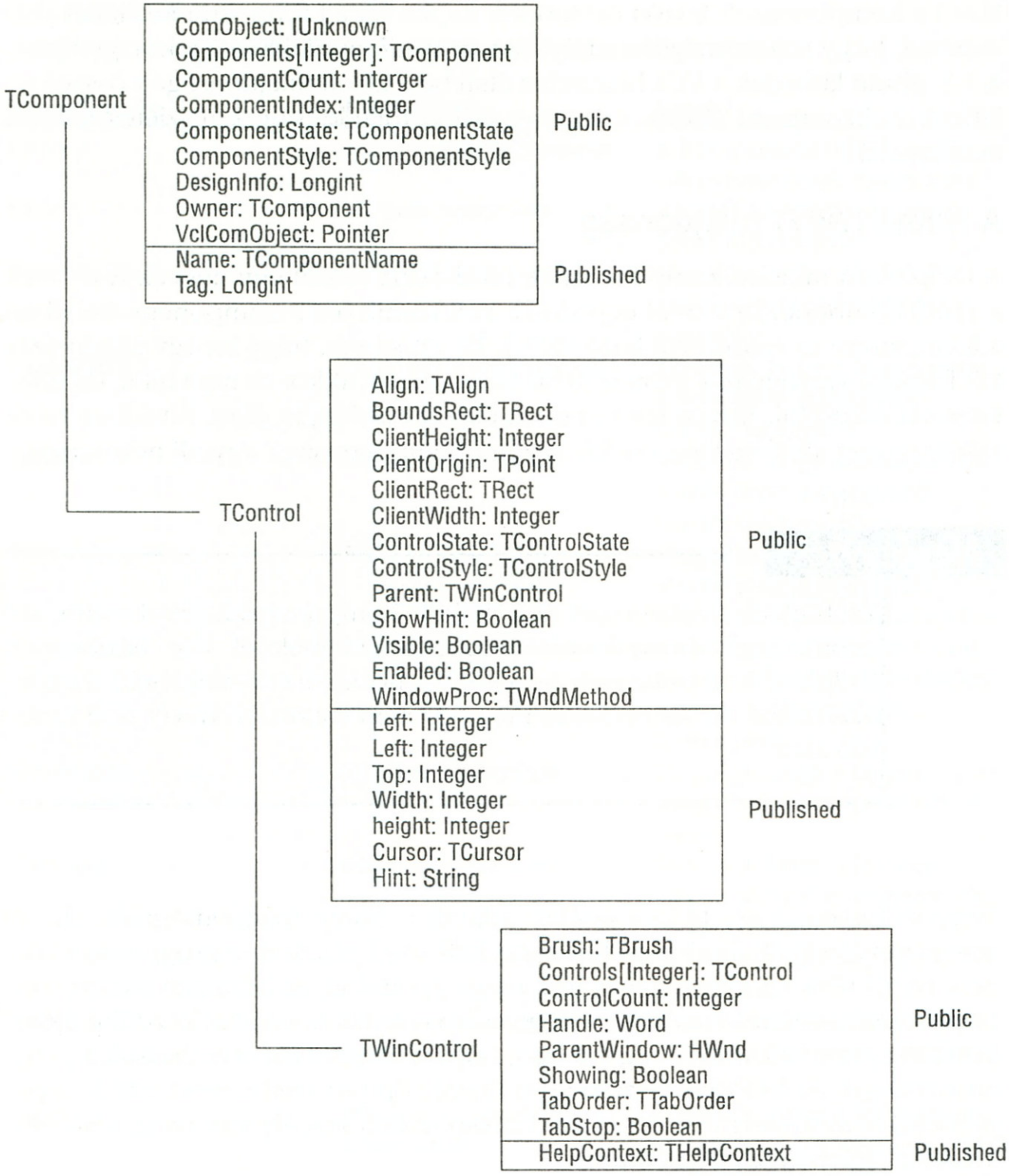
```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  for I := 0 to ComponentCount - 1 do
    ListBox1.Items.Add (Components [I].Name);
end;
```

Ez a program a ComponentCount (ez a formon lévő komponensek számát tartalmazza) és a Components (ez a komponensek listáját tartalmazzó tömb) tulajdonságokat használja. A Components tulajdonság által tartalmazott lista TComponent típusú értékeket tartalmaz, így azokra közvetlenül alkalmazhatjuk a Name tulajdonságot. Ha más, típus-specifikus tulajdonságokat is használni szeretnénk, akkor explicit típuskonverziót kell használnunk, az as kulcsszó segítségével.

Megjegyzés

A Delphiben létezik néhány komponens, amelyek maguk is komponenstárolók: ilyen a GroupBox, a Panel, a Windows 95 Page Control, a Notebook, a TabbedBotebook és természetesen a Form komponens. Ha ezeket az összetevőket használjuk, akkor elhelyezhetünk további komponenseket bennük. Ebben az esetben az elhelyezett elemek szülője a tároló lesz (ahogy azt a Parent tulajdonság is jelzi), tulajdonosuk viszont a form (ahogy az Owner tulajdonság mutatja). A gyermek kontrolok irányítására a Control tulajdonságot használhatjuk, a Components tulajdonsággal pedig az összes, a form tulajdonában lévő összetevőt, függetlenül annak szülőjétől.

A Component tulajdonság használatával mindig minden formon található komponenshez hozzáférhetünk. Ha azonban egy konkrét elemet keresünk a listában, akkor ahelyett, hogy a lista összes elemével összehasonlítanánk a keresettet, használhatjuk a Delphi FindComponent metódusát.



7.7. ábra *A tulajdonságok, amik a VCL hierarchia legfelsőbb szintjén kerülnek bevezetésre, elérhetők minden alosztály számára*

- Harmadszor, a Delphi a komponens nevéből fogja származtatni az ahhoz kapcsolódó események nevét. Ha van például egy Button1 gombunk, akkor a megnyomásából származó esemény kezelőjének neve Button1Click lesz, hacsak konkrétan meg nem adunk neki egy másik

nevet. Ha később megváltoztatjuk a komponens nevét, akkor a hozzá tartozó eseménykezelő neve is automatikusan megváltozik. Ha az előbbi példában a gomb nevét MyButton-ra változtatjuk, akkor az eljárás neve is Button1Click-ről MyButtonClick-re változik.

Mindenesetre ne feledjük, hogy amellett, hogy név szerint hivatkozunk egy komponensre, használhatjuk a hozzá tartozó form Components és Controls tulajdonságát is. Itt egy példaprogram, amiben ezek használatát mutatjuk be:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  for I := 0 to ComponentCount - 1 do
    ListBox1.Items.Add (Components [I].Name);
end;
```

Ez a program a ComponentCount (ez a formon lévő komponensek számát tartalmazza) és a Components (ez a komponensek listáját tartalmazzó tömb) tulajdonságokat használja. A Components tulajdonság által tartalmazott lista TComponent típusú értékeket tartalmaz, így azokra közvetlenül alkalmazhatjuk a Name tulajdonságot. Ha más, típus-specifikus tulajdonságokat is használni szeretnénk, akkor explicit típuskonverziót kell használnunk, az as kulcsszó segítségével.

Megjegyzés

A Delphiben létezik néhány komponens, amelyek maguk is komponenstárolók: ilyen a GroupBox, a Panel, a Windows 95 Page Control, a Notebook, a TabbedBotebook és természetesen a Form komponens. Ha ezeket az összetevőket használjuk, akkor elhelyezhetünk további komponenseket bennük. Ebben az esetben az elhelyezett elemek szülője a tároló lesz (ahogy azt a Parent tulajdonság is jelzi), tulajdonosuk viszont a form (ahogy az Owner tulajdonság mutatja). A gyermek kontrolok irányítására a Control tulajdonságot használhatjuk, a Components tulajdonsággal pedig az összes, a form tulajdonában lévő összetevőt, függetlenül annak szülőjétől.

A Component tulajdonság használatával mindig minden formon található komponenshez hozzáférhetünk. Ha azonban egy konkrét elemet keresünk a listában, akkor ahelyett, hogy a lista összes elemével összehasonlítanánk a keresettet, használhatjuk a Delphi FindComponent metódusát.

Komponensek méretére és helyzetére vonatkozó tulajdonságok

A tulajdonságok másik fontos csoportjába tartoznak a komponens helyzetét és méretét meghatározók. Az összetevő formon belüli helyét a Left (Bal) és Top (Teteje) tulajdonságok, méretét a Height (Magasság) és Width (Szélesség) tulajdonságok adják meg. A rendszer még a futás közben nem látható elemeknek is tárolja a pozícióját, így a form újabb megnyitásakor ugyanott lesznek, ahová utoljára helyeztük őket.

A pozíciók fontos tulajdonsága, hogy azok mindig relatívak a komponens szülőjének kliens területéhez (amit a komponens Parent (Szülő) tulajdonsága mutat), ugyanúgy, ahogy általában a Windows-ban. Egy formnál a kliens terület az ablak keret és fejléc nélküli részét jelenti. Noha igen nyűgös dolog abszolút képernyő-koordinátákkal dolgozni, léteznek olyan metódusok, amik az abszolút és relatív koordinátákat konvertálják oda és vissza.

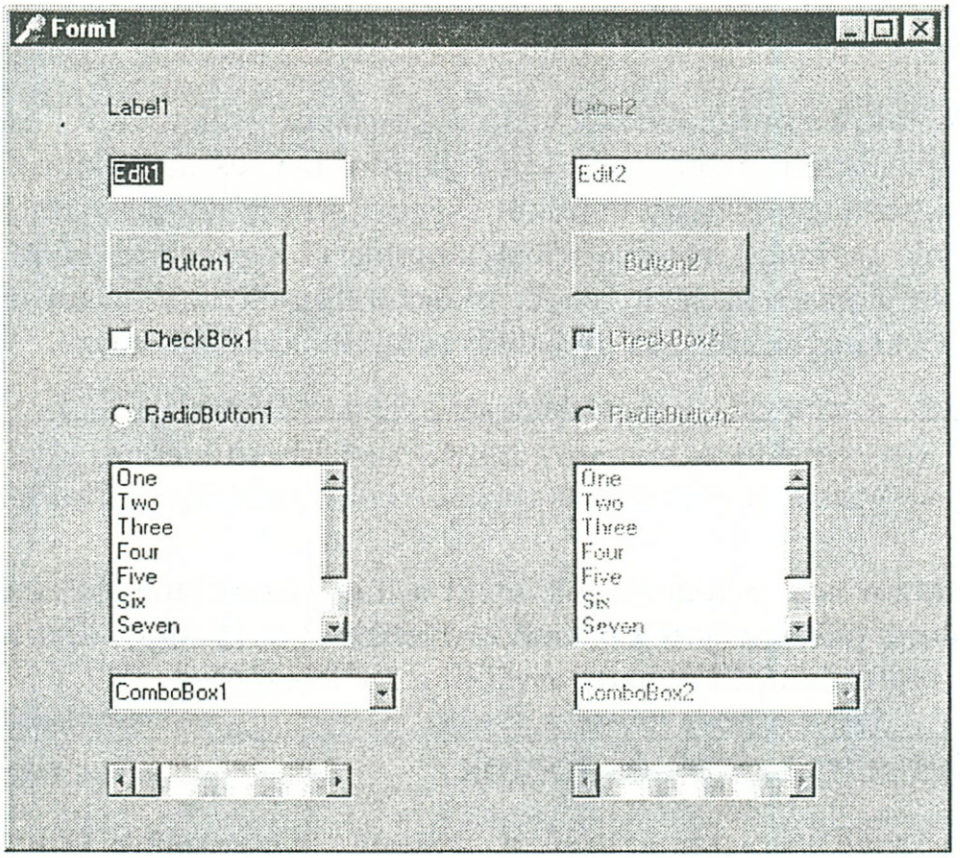
Fontos megjegyeznünk, hogy a komponensek koordinátái lehetnek egy panelhez, vagy egy úgynevezett konténer (container) komponenshez is relatívak. Ha a formunkra teszünk például egy panelt, és abban elhelyezünk egy gombot, akkor annak helyzete nem a formhoz, hanem a panelhez lesz viszonyítva, hiszen az lesz a gomb szülője.

Aktivizálási és láthatósági tulajdonságok

Két alapvető tulajdonság van, amik segítségével a felhasználó aktiválhatja, vagy elrejtheti a komponenseket, ezek közül a legegyszerűbb az Enabled (Engedélyezett) tulajdonság. Ha egy komponens le van tiltva (vagyis az Enabled tulajdonság False-ra (hamisra) van állítva), akkor azt a rendszer valamilyen módon tudtára adja a felhasználónak. Tervezési időben a letiltásnak nem mindig van hatása az összetevőre, de futási időben ez mindig jelölve van valahogy, általában a komponens beszürkítésével, ahogy azt a 7.8. ábrán is láthatjuk. Ha ki akarjuk próbálni, hogy hogyan működnek a letiltott összetevők akkor futtassuk le a mellékelt CD-n található Disabled című programot.

Tipp

Ha le akarjuk tiltani a jobb oldali oszlop minden kontrolját, akkor egyszerűen válasszuk ki mindet a SHIFT segítségével, és állítsuk az Enabled tulajdonságot egyszerre az összes kiválasztott elemnél.



7.8. ábra *Az engedélyezett (balra) és a letiltott (jobbra) kontrolok felhasználói felületen*

Egy, az előbbinél radikálisabb módszer az, amikor teljesen elrejtjük a komponenst a megfelelő Hide (Elrejt) metódussal, vagy a Visible (Látható) tulajdonság hamisra állításával. Ne feledjük azonban, hogy a Visible tulajdonság nem mondja meg, hogy az összetevő látható-e vagy sem, előfordulhat, hogy hiába állítjuk ezt igazra, mert ha például a komponens konténere rejtve van, akkor az összetevő sem fog látszani. Éppen ezért létezik a Showing (Látszik) futásidejű, csak olvasható tulajdonság, ami ha igaz, akkor az azt jelenti, hogy a komponens, annak szülője, a szülő szülője stb. mind láthatóak.

A testreszabható Tag tulajdonság

Ez a tulajdonság igen furcsának tűnhet, hiszen az égvilágon semmi hatással nincs semmire. A Tag nem más, mint egy extra érték a memóriában, amely minden komponensnél megtalálható, és amelyben egy tetszőleges értéket tárolhatunk. Az, hogy milyen típusú információt tárolunk itt, teljesen tőlünk függ.

Ez az eszköz nagyon hasznos lehet, ha valami plusz információt szeretnénk a komponenseinkkel tárolni, hiszen így ezt annak újradefiniálása nélkül is megtehetjük. Technikai szempontból a Tag egy hosszú egész számot (long integer) tárol, így gyakorlatilag bármit (például egy pointert, vagy egy tömbindexet) beleírhatunk, ami négy bájt hosszú.

Megjegyzés

Tapasztalt Windows programozók talán rokonságot vélhetnek felfedezni a Delphi komponenseinek Tag tulajdonsága, és az ablakok extra bájtja között. Igaz, hogy mindkét eszközt arra lehet használni, hogy általunk definiált információval lássunk el egy előre adott struktúrát, de a hasonlóság itt véget is ér. A Tag-ek használata a Delphiben jóval egyszerűbb és gyorsabb is, mint az extra bájtoké.

A későbbi programokban sok példát látunk majd a Tag használatára. Ezek közé tartozik a Numbers példa a 8. fejezetben, a RichNot2, a BitMenu és az ODMenu példák a 9. fejezetben, és néhány további a 25. fejezetben.

A felhasználói interfész: színek és fontok

A felhasználói felület beállítására két legtöbbet használt tulajdonság a Color (Szín) és a Font (Betűtípus). A szín megadásához általában több különböző értéket is beállíthatunk, maga a Color általában az adott komponens háttérszínére vonatkozik. Sok komponensnek van ParentColor (SzülőSzín) és ParentFont (SzülőFont) tulajdonsága, amik azt jelzik, hogy a komponensek a szülőjük színét örököljék, vagy sem. Ezeket a tulajdonságokat könnyedén használhatjuk például arra, hogy az egész form minden összetevőjének megváltoztassuk a betűtípusát úgy, hogy csak a formét állítjuk át, a többiét pedig a szülőtől (ami jelen esetben maga a form) örököltetjük.

Mikor az Object Inspector megfelelő mezőjébe beírjuk a használni kívánt betűtípust, vagy mikor egy standard font-kiválasztó dialógusablakot használunk, mindig a számítógépre telepített fontok közül választunk egyet. A ténynek, hogy a Delphi lehetővé teszi az összes számítógépre installált betűkészlet használatát, van előnye is és hátránya is. Az előny az, hogy ha sok szép betűtípusunk van, akkor tetszetős programokat készíthetünk, a hátrány viszont abban áll, hogy ha terjesztjük programunkat, akkor egyáltalán nem biztos, hogy az általunk használt fontkészlet megtalálható lesz a másik fél számítógépén is.

Ha a programunk olyan betűkészletet használ, ami nincs a számítógépen telepítve, akkor a Windows helyettesíti azt egy másikkal. Ez sajnos sok esetben teljesen tönkreteheti a program gondosan megtervezett felhasználói felületét. Erre a problémára két megoldás kínálkozik: az egyik a Windows standard betűkészleteinek (az MS Sans Serif, a System, az Arial, a Times New Roman és társaik) használata, a másik pedig az, hogy szoftverünkkel együtt szállítjuk az általa használt fontokat is, feltéve, hogy azok licenszszerződése ezt lehetővé teszi.

Egy szín értékét többféleképpen is be lehet állítani. Ennek a tulajdonságnak a típusa a TColor. Ha ilyen típusú tulajdonságot akarunk megadni, akkor választhatunk a nyelv előre definiált konstansai közül, vagy beírhatjuk a konkrét értéket. A konstansok a következők:

```
clAqua      clBlack     clBlue      clDkGray
clFuchsia   clGray      clGreen     clLime
clLtGray    clMaroon    clNavy      clOlive
clPurple    clRed       clSilver    clTeal
clWhite     clYellow
```

Még jobb megoldásként használhatjuk a Windows rendszerösszetevőinek (mint például az ablakok háttere, vagy az aktív felirat stb.) színeit. A 7.2. táblázat az ide vonatkozó konstansok listáját tartalmazza.

7.2. táblázat

A Windows rendszerösszetevői által használt színek

Színkonstans	Minek a színét tartalmazza?
cl3DdkShadow	A három dimenziós hatások sötét árnyék effektje
cl3Dlight	A három dimenziós hatások világos effektje
clActiveBorder	Az aktív ablak határa
clActiveCaption	Az aktív ablak címsora
clAppWorkSpace	Az alkalmazás munkaterülete
clBackground	A Windows háttere
clBtnFace	A gombok felülete
clBtnHighlight	A kijelölt gomb felülete
clBtnShadow	A gomb által vetett árnyék
clBtnText	A gombon olvasható szöveg
clCaptionText	Az aktív ablak címsorában olvasható szöveg
clGrayText	Az elszürkített szöveg
clHighlight	A kijelölt szöveg háttere
clHightlightText	A kijelölt szöveg
clInactiveBorder	Az inaktív ablakok határa
clInactiveCaption	Az inaktív ablak címsora
clInactiveCaptionText	Az inaktív ablak címsorának szövege
clInfoBk	A tipp ablak háttere
clInfoText	A tipp ablak szövege
clMenu	A menük háttere
clMenuText	A menüelemek szövege
clScrollBar	A gördítősávok
clWindow	Az ablakok háttere
clWindowFrame	Az ablakok kerete
clWindowText	Az ablakok szövege

A TColor típusú tulajdonságokat úgy is értéket beállíthatjuk, hogy az előre definiált konstansok helyett egy négybájtos, hexadecimális számot adunk meg nekik. Ha ezt a módszert alkalmazzuk, akkor tudnunk kell, hogy ezek a számok az RGB színintenzitások beállítását jelentik; a szám alsó három bájtja a kék, a zöld, és a vörös szín erősségét határozza meg. Így például a $00FF0000 a kék, a $0000FF a zöld, a $000000FF pedig a vörös színt jelentik, a $00000000-val feketére, a $00FFFFFF-el fehérre állíthatjuk a megjelenítendő színt. Ezzel a módszerrel 16 millió különböző színárnyalat közül válogathatunk.

A hexadecimális értékek közvetlen használata helyett személy szerint inkább az RGB függvényt ajánlom. Ennek három, 0 és 255 közötti paramétere van, az első a vörös, a második a zöld, a harmadik a kék színre vonatkozik. A függvény használata leegyszerűsíti munkánkat, és olvashatóbbá tesz programunkat is.

Megjegyzés

Az RGB *majdnem* egy Windows API függvény. A Windowsra, és nem a Delphire vonatkozó egységekben van definiálva, de hasonló függvény nem létezik a Windows API-ban. A C nyelvben egy makrónak van hasonló neve és hatása, így a Pascal interfésznek ez egy igen kellemes kibővítése.

A TColor típus legmagasabb helyértékű bájtja azt mondja meg, hogy a Windows melyik palettában keresse meg a megadott színhez legközelebb lévőt, de arról, hogy ez pontosan mit jelent, nem lesz szó ebben a könyvben. Ezt a bájtot egyébként különböző grafikai programok áttetszőségi információ tárolására is szokták használni. A paletták és színek kapcsán meg kell jegyeznünk, hogy a Windows néha helyettesít néhány színt az ahhoz legközelebb álló, általa megjeleníthetővel, legalábbis olyan VGA üzemmódoknál, amelyek palettákat használnak. A fontoknál, vonalaknál, és hasonlóknál mindig ez a helyzet. Más esetekben a rendszer megpróbálhatja „kikeverni" a rendelkezésére álló színekből a megjelenítendőket, ami igen furcsa eredményekhez vezethet. 16-színű (VGA) adaptereknél, vagy nagy képfelbontásnál könnyen úgy járhatunk, hogy a kívánt színünk helyett csak valami összevisszaságnak tűnő pixelmintázat jelenik meg ott, ahol homogén színt szerettünk volna látni.

Általános VCL metódusok

A komponens metódusok ugyanolyanok, mint bármelyik másik, korábban megismert metódus: vannak eljárásaink és függvényeink, melyeket meghívhatunk a megfelelő időben. Ahogy azt már korábban említettem, egy metódust

sokszor arra is lehet használni, hogy egy tulajdonságot írjunk vagy olvassunk. A tulajdonságok használatával érthetőbbé, könnyebben olvashatóbbá tudjuk tenni programunkat, de a metódusok nem csak erre jók, sőt, legtöbbjüknek nincs is hozzá tartozó tulajdonsága. Sok közülük egy eljárás, ami végrehajt egy műveletet, és nem egy értéket ír, vagy olvas. Vannak metódusok, amik csak a komponensek, míg mások csak a kontrolok (vizuális komponensek) által érhetők el. A 7.3. táblázat a legtöbbször használt metódusokat tartalmazza; a könyvben később példákat is látunk majd ezek használatára.

7.3. táblázat

Néhány, a legtöbb VCL komponens által elérhető metódus

Metódus	Elérhető	Leírás
BeginDrag	Minden kontrolnál	Elindítja a manuális húzást az egérrel.
BringToFront	Minden kontrolnál	A kontrolt a többi elé helyezi.
CanFocus	Minden kontrolnál	Megadja, hogy a kontrol fókuszba kerülhet-e.
ClientToScreen	Minden kontrolnál	A kliens koordinátákat képernyő-koordinátákká alakítja.
ContainsControl	Minden kontrolnál	Megadja, hogy egy bizonyos kontrolt tartalmaz-e az aktuális.
Create	Minden komponensnél	Egy új példányt készít (konstruktor).
Destroy	Minden komponensnél	Megsemmisít egy példányt (destruktor). Itt valójában a Free-t kellene meghívnunk.
Dragging	Minden kontrolnál	Megadja, hogy a kontrolt húzzák-e éppen valahova az egérrel.
EndDrag	Minden kontrolnál	Manuálisan leállítja az egérrel való húzást.
FindComponent	Minden komponensnél	Visszaadja a komponenstömb megadott nevű tagját (ezt használtuk a NameProp példában).
Focused	Minden ablak alapú kontrolnál	Megadja, hogy a kontrol a fókuszban van-e.
Free	Minden komponensnél	Törli az objektumot a memóriából (formoknál a Release metódust kell használni).
GetTextBuf	Minden kontrolnál	Visszaadja a kontrol szövegét, vagy feliratát.

7.3. táblázat

Néhány, a legtöbb VCL komponens által elérhető metódus

Metódus	Elérhető	Leírás
GetTextLen	Minden kontrolnál	Visszaadja a kontrol szövegének, vagy feliratának hosszát.
HandleAllocated	Minden kontrolnál	Igazat ad vissza, ha egy rendszer-kezelő (system handler) le lett foglalva a kontrol számára.
HandleNeeded	Minden kontrolnál	Lefoglal egy megfelelő rendszer-kezelőt, ha az nem létezik.
Hide	Minden kontrolnál	Láthatatlanná teszi a kontrolt (ugyanaz, mintha a Visible tulajdonságot hamisra állítanánk).
InsertComponent	Minden komponensnél	Beszúr egy új elemet a tulajdonban levő komponensek lsitájába.
InsertControl	Minden kontrolnál	Egy új kontrolt szúr be az aktuális gyermekeinek listájába.
Invalidate	Minden kontrolnál	Kikényszeríti a kontrol újrarajzolását.
RemoveComponent	Minden komponensnél	Eltávolítja a komponenst a komponens-listából.
ScaleBy	Minden kontrolnál	Átméretezi a kontrolt egy adott százalékkal.
ScreenToClient	Minden kontrolnál	Képernyő-koordinátákat kliens koordinátákká alakít.
ScrollBy	Minden kontrolnál	Görgeti a kontrol tartalmát.
SendToBack	Minden kontrolnál	Hátraküldi a kontrolt.
SetBounds	Minden kontrolnál	A kontrol helyzetét és méretét változtatja (ez így gyorsabb módszer, mint a megfelelő tulajdonságok egyenkénti átállítása).
SetFocus	Minden kontrolnál	A kontrolnak adja az input fókuszt.
SetTextBuf	Minden kontrolnál	Beállítja a kontrol szövegét (vagy feliratát).
Show	Minden kontrolnál	Láthatóvá teszi a kontrolt (ugyanaz, mintha a Visible tulajddnságot igazra állítanánk).
Update	Minden kontrolnál	Azonnal újrafesti a kontrolt.

Az általános VCL események

Éppen úgy, ahogy léteznek minden komponenssel használható tulajdonságok, ugyanúgy léteznek olyan események, amelyek mindegyikük számára elérhető. A 7.4. táblázat rövid leírást tartalmaz ezekről az eseményekről. Nem győzöm azonban hangsúlyozni, hogy ez csupán bevezetés ebbe a témába, a könyv későbbi fejezeteiben még bőven látunk majd példákat ezek használatára.

7.4. táblázat

Néhány, minden komponens által elérhető esemény

Metódus	Elérhető	Leírás
OnChange	Sok komponensnél	Akkor történik, mikor egy objektum, vagy annak valamilyen adata megváltozik.
OnClick	Legtöbb kontrolnál	Akkor történik, mikor az egér bal gombjával kattintunk a komponensen.
OnDblClick	Sok kontrolnál	Akkor történik, mikor az egér bal gombjával duplán kattintunk a komponensen.
OnDragDrop	Legtöbb kontrolnál	Akkor történik, mikor egy húzás művelet a kontrol felett ér véget; ezt az eseményt az a komponens generálja, amelyik a húzás művelet eseményét kapta.
OnDragOver	Legtöbb kontrolnál	Akkor történik, mikor a kontrol fölé húzzuk az egeret.
OnEndDrag	Legtöbb kontrolnál	Akkor történik, mikor egy húzás művelet véget ér; ezt az eseményt az a komponens generálja, amelyik a húzás művelet eseményét kapta.
OnEnter	Minden ablak alapú kontrolnál	Akkor történik, mikor a komponens aktivizálódik, vagyis megkapja a fókuszt.
OnExit	Minden ablak alapú kontrolnál	Akkor történik, mikor a komponens elveszíti a fókuszt.
OnKeyDown	Néhány ablak alapú kontrolnál	Akkor történik, mikor a felhasználó lenyom egy billentyűt, és az a fókuszban levő komponenshez továbbítódik.

7.4. táblázat

Néhány, minden komponens által elérhető esemény

Metódus	Elérhető	Leírás
OnKeyPress	Néhány ablak alapú kontrolnál	Akkor történik, mikor a felhasználó leüt egy billentyűt, és az a fókuszban levő komponenshez továbbítódik.
OnKeyUp	Néhány ablak alapú kontrolnál	Akkor történik, mikor a felhasználó felenged egy billentyűt, és az a fókuszban levő komponenshez továbbítódik.
OnMouseDown	Legtöbb kontrolnál	Akkor történik, mikor a felhasználó lenyom egy egérgombot; az esemény az egérmutató alatt levő komponenshez továbbítódik.
OnMouseMove	Legtöbb kontrolnál	Akkor történik, mikor az egérmutató a komponens fölé kerül; az esemény az egérmutató alatt levő komponenshez továbbítódik.
OnMouseUp	Legtöbb kontrolnál	Akkor történik, mikor a felhasználó felenged egy egérgombot; az esemény az egérmutató alatt levő komponenshez továbbítódik.
OnStartDrag	Legtöbb kontrolnál	Akkor történik, mikor a felhasználó elkezd egy komponenst húzni az egérrel; az esemény a művelet kiinduló komponenséhez továbbítódik.

A Delphi kollekcióinak használata

A különböző nem-komponens osztályok között az egyik legfontosabb a kollekciók csoportja. Alapvetően háromfajta kollekció osztály található a Delphiben:

- A TList egy pointerlistát definiál, amelyben bármilyen osztályú objektumokat tárolhatunk. A TList sokkal rugalmasabb a tömböknél, mert mérete futásidőben is változhat.

- A TStrings egy absztrakt osztály, ami mindenféle típusú string listák reprezentálására szolgál, azok tárolási módjától függetlenül. Ez az osztály egy absztrakt string listát definiál, ezért csak olyan komponensek tulajdon-ságainál használhatjuk őket, amik maguk is képesek stringeket tárolni (ilyen például a listbox).
- A TStringList a TStrings egy alosztálya, egy string listát definiál, melynek van saját tároló helye. Ezt az eszközt használhatjuk programjainkban saját listáink elkészítésére.
- A TCollection egy homogén objektumlistát definiál, melynek a kollekció osztály a tulajdonosa. A lista elemeinek a TCollectionItem osztály leszármazottjainak kell lenniük. Ha adott objektumok tárolására alkalmas kollekcióra van szükségünk, akkor azt mind a TCollection, mind a TCollectionItem alosztályként elkészíthetjük.

A Borland programozói úgy döntöttek, hogy ezeket az osztályokat listáknak fogják elnevezni, és én is követni fogom ezt a konvenciót, annak ellenére, hogy ezek dinamikus tömbök, és nem hagyományos láncolt listák. A listák mindegyikének vannak hozzájuk tartozó metódusai és tulajdonságai. A listák műveleteit a szokásos, tömbös jelöléssel („[" és „]") végezhetjük, akár kiolvasni, akár cserélni akarunk egy elemet. A tulajdonságok közé tartozik a Count (számlál), és tartoznak hozzájuk olyan tipikus metódusok, mint az Add (hozzáad), a Delete (töröl), az Insert (beszúr), a Remove (eltávolít), és a keresőműveletek (például az IndexOf).

Vegyük észre, hogy a TStringList és a TStrings objektumok stringlistákat és objektumlistákat is hozzárendelhetnek stringekhez. Ez a tulajdonság egy rakás új felhasználási lehetőséget ad a kezünkbe: használhatjuk például őket hozzárendelt objektumok „szótáraként", vagy bittérképeket és egyéb objektumokat tárolhatunk bennük, hogy később egy list boxban felhasználjuk azokat.

Megjegyzés

A TListbox komponens konkrétan egy TStringList objektumot használ, amíg az érvénytelen ablakleírója miatt stringeket kell tárolnia, és egy TStrings objektumot használ, mikor végül hozzárendeljük a Windows list box kontroljához, amely a saját karakterláncait tárolja.

A két stringlista osztálynak szintén van néhány jól használható eljárása, a SaveToFile és a LoadFromFile, amelyek segítségével szövegfájlba menthetjük, és onnan visszatölthetjük tartalmukat. Ha iterációs lépéseket akarunk a listán végezni, akkor egyszerűen írjunk egy, az indexükön alapuló ciklust, vagy használjuk a List tömbtulajdonságot.

Objektum- és adatlisták használata

Készítsünk most egy példát az eddigiekre, az általános TList osztály használatával. Ha tetszőleges típusú objektumokból szeretnénk listát készíteni, akkor annyit kell tennünk, hogy deklarálunk egy TList objektumot, feltöltjük az adatokkal, majd az adatok eléréséhez a megfelelő típusúra konvertáljuk őket. A ListDemo példa éppen ezt illusztrálja, de bemutatja ennek a megközelítésnek néhány csapdáját is. A program formja két private típusú változót tartalmaz, melyek két listát tárolnak:

```
private
  ListNum, ListDate: TList;
```

A form létrejöttekor a következő listaobjektumokat készítjük el:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  ListNum := TList.Create;
  ListDate := TList.Create;
end;
```

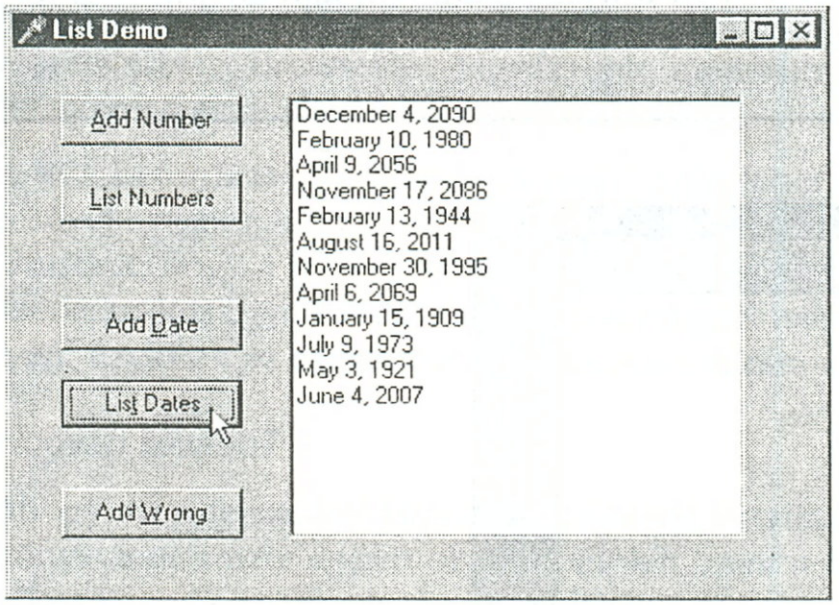
A form két gombja egy-egy véletlen számot és dátumot ad a megfelelő listához (természetesen a projekt tartalmazza a dátumot leíró egységet is):

```
procedure TForm1.ButtonAddNumClick(Sender: TObject);
begin
  ListNum.Add (Pointer (Random (10000)));
end;
procedure TForm1.ButtonAddDateClick(Sender: TObject);
begin
  ListDate.Add (TDate.Init (1 + Random (12),
    1 + Random (31), 1900 + Random (200)));
end;
```

Vegyük észre, hogy a TDate komponens használatakor az Init hívás kivételt válthat ki, ha az adott értékek egy érvénytelen dátumot adnának meg. Mikor a listából kivesszük az elemeket, vissza kell őket konvertálni az eredeti típusukra, ahogy azt a következő két metódusban is látni fogjuk (a második működését a 7.9. ábra szemlélteti):

```
procedure TForm1.ButtonListNumClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to ListNum.Count - 1 do
    Listbox1.Items.Add (IntToStr (Integer
        (ListNum[I])));
end;
```

```
procedure TForm1.ButtonListDateClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to ListDate.Count - 1 do
    Listbox1.Items.Add ((
      TObject(ListDate [I]) as TDate).GetText);
end;
```



7.9. ábra *A ListDemo példa által mutatott dátumok listája*

A fenti kód végén a TList által visszaadott pointert egy TObject referenciává kell alakítani, különben nem vihető végbe az as művelet. Ez a fajta művelet egyébként „érvénytelen típuskonverzió kivételt" (invalid typecast exception) válthat ki, vagy memóriahibát eredményezhet, ha a pointer nem egy objektumra mutat.

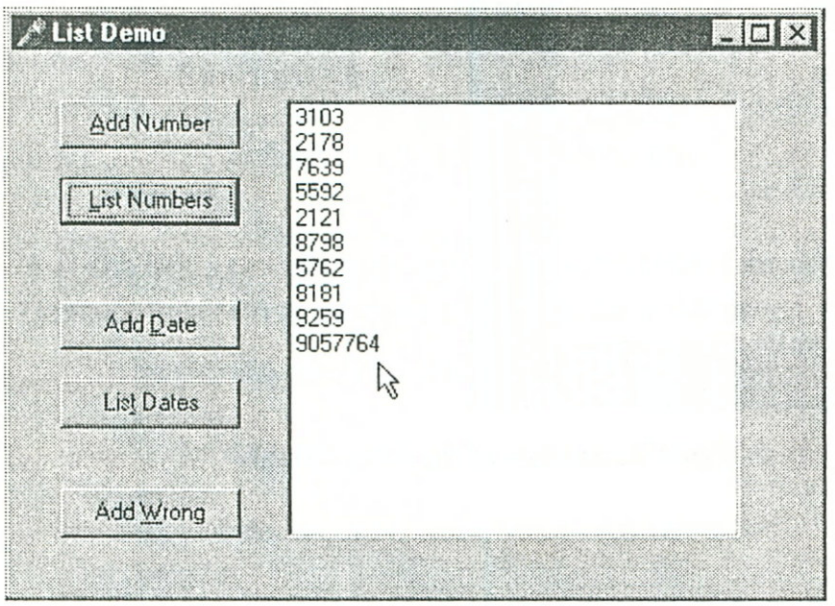
Hozzáadtam a formhoz még egy gombot, hogy megmutassam, hogy a dolgok bizony tényleg elromolhatnak a példában. Ez a gomb lenyomásakor a TButton objektumot adja hozzá mindkét listához, vagyis azt a gombot, amit épp akkor nyomtunk meg:

```
procedure TForm1.ButtonWrongClick(Sender: TObject);
begin
  // add a button to both lists
  ListNum.Add (Sender);
  ListDate.Add (Sender);
end;
```

Ha megnyomjuk a gombot és frissítjük a listák valamelyikét, akkor hibaüzenetet kapunk. A dátumlista kivált egy kivételt, a számok listája pedig számnak fogja hinni az objektum címét, ahogy azt a 7.10. ábrán láthatjuk.

Figyelem!

Mikor megsemmisítünk egy objektumlistát, soha ne felejtsük el először a lista elemeit megsemmisíteni. Ezt is elvégeztem a form FormDestroy metódusában, mert a listaobjektumok karbantartása és felszabadítása nem automatikus a Delphiben.



7.10. ábra *Ha egy objektumot adunk a számok listájához a ListDemo példában, akkor valójában a címét kapjuk vissza*

A VCL forráskód tanulmányozása

Ebben a fejezetben éppen csak belekóstoltunk a VCL rejtelmeibe, de ne aggódjunk, még sok mindent meg fogunk róla tanulni, ahogy haladunk tovább a könyvben. A részletesen meg akarjuk ismerni a komponenseket, akkor erre létezik egy másik mód is: a VCL forráskódja megtalálható a Delphi Professional és Client/Server Suite verzióiban.

A Delphi ezen változatai a futásidejű (RTL) és a VCL könyvtárak forráskódjával együtt kaphatók. A VCL forrásszövegének tanulmányozása nagy segítség lehet abban az esetben, ha az új komponensek készítésének, és a már meglévők

kibővítésének titkait akarjuk ellesni, de ne próbáljuk meg innen megérteni, hogyan működnek a meglévő komponensek, ugyanis ehhez tisztában kell lennünk a Delphi és a Windows programozás csínjával-bínjával.

Ha sok időt akarunk a Delphi forráskódjának tanulmányozásával tölteni, akkor azt ajánlom, hogy szerkesztőprogram helyett használjuk inkább a beépített debuggert, vagy az Object Browsert. Ahhoz, hogy ezek az eszközök lássák a VCL forráskódját, annak elérési útját (általában `C:\Program Files\Borland\Delphi 3.0\Source\VCL`) a Project Options párbeszédablak **Directories** oldalának **Search Path** pontjánál állítsuk be. Ha ezután lefordítunk egy projektet, akkor a VCL forrás is fordításra kerül, és minden információ bekerül róla az Object Browserbe. Vigyázzunk, mert ehhez a művelethez bekapcsolt hibakereső és jó nagy adag memória szükséges!

A VCL befordítása forráskódunk mellé azért lehet érdekes, mert így a hibakeresővel nyomon követhetjük annak működését. Az Object Browsert használva láthatjuk, hogy a forráskódban hol, melyik osztályra történik hivatkozás, és a szerkesztőt is egyetlen egérkattintással megnyithatjuk a számunkra érdekes helyen. Ezeknek az eszközöknek a részletes leírása a 26. fejezetben szerepel.

Hogyan tovább?

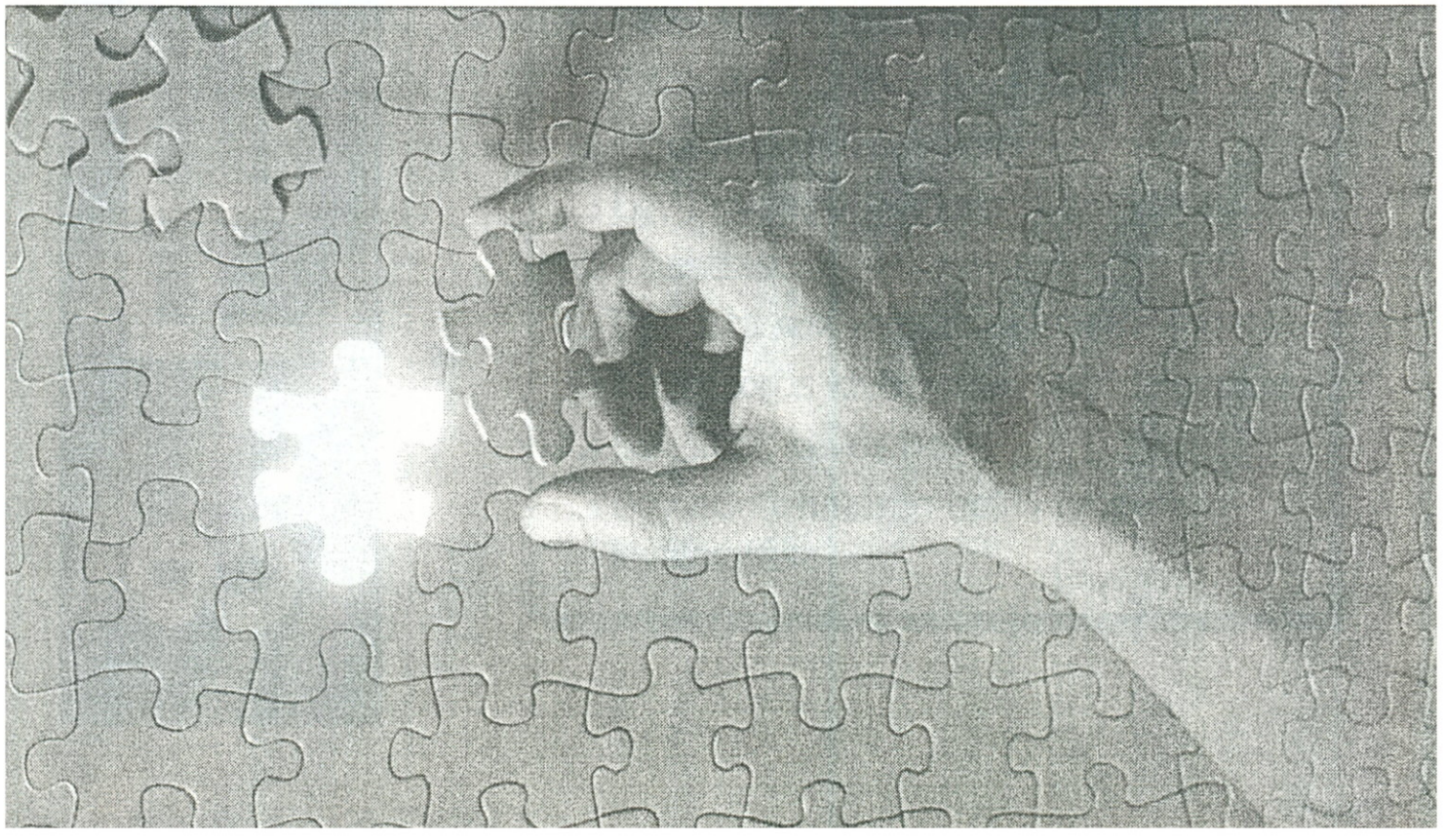
Ahogy azt a fejezetből láthattuk, a Delphi tartalmaz egy teljes értékű osztálykönyvtárat, amely éppen olyan komplett, mint a Microsoft MFC C++ osztálykönyvtára. Természetesen a Delphi VCL-e sokkal objektumorientáltabb, és sokkal magasabb szintű absztrakciós képességekkel rendelkezik a Windows API felett, mint a C++ könyvtárak.

A komponensek használatához csupán a VCL hierarchia legtetején ülő elemek ismerete szükséges; ezek nagyjából azok a komponensek, amelyek a komponenspalettán megtalálhatók. Igazából ezek használatához nincs szükség a VCL átfogóbb ismeretére, csak akkor, ha új komponenseket akarunk készíteni, vagy a régieket módosítani.

Ezzel a fejezettel véget ér a könyv I. része, amely a Delphi programozás alapjaival foglalkozott. A II. részt teljes egészében a különböző komponensek bemutatásának szenteltem: a 8. fejezetben a hagyományos Windows kontrolokkal kezdünk, a 9. fejezetben áttérünk a menükre, a 10. fejezetben a TForm komponenssel, a grafikus kimenettel, majd a grafikus komponensekkel, eszköztárakkal, státuszsorokkal, dialógusablakokkal, MDI alkalmazásokkal, és végül az adatbázis-kezeléssel kapcsolatos komponensekkel ismerkedünk meg.

II. Rész

Komponensek használata



8. FEJEZET

Alapvető komponenesek

E fejezet tartalma:

- A Windows saját komponensei
- Gombra kattintás
- Kattintás az egérrel
- Színes szövegek a formon
- Elemek áthúzása komponensek között
- Felhasználói input fogadása
- Kifinomult beviteli sémák
- Egy egyszerű szerkesztő készítése
- Választások
- Egy lista sok választási lehetőséggel
- Többszörös kiválasztás engedélyezése
- Sok lista, kevés hely
- Egy adott határon belüli értékek kiválasztása

Alapvető komponensek

Most, hogy bemutattuk a Delphi környezetet és áttekintettük az Object Pascal nyelvet a Visual Component Libraryval együtt, belemélyedhetünk a könyv központi részébe: a komponensek használatába. A Delphi igazából erről szól, fejlesztői környezetének kulcsfontosságú tulajdonsága az komponensek segítségével történő, az objektumokon alapuló programozás.

A rendszer számos használatra kész komponenssel rendelkezik. Egyenként nem fogok minden komponenst részletezni tulajdonságaival és eljárásaival együtt, az efféle információk könnyen megtalálhatók a Súgó rendszerben. A könyv második részének célja azt bemutatni, hogyan lehet a Delphi előre megadott komponensei által nyújtott lehetőségeket bizonyos alkalmazások elkészítésére használni. Ez, és a következő fejezetek igazából példaprogramokon alapulnak. Ezek a példák szándékosan nagyon egyszerűek – bár a lehetőségekhez mérten életszerűek –, hogy egyszerre csak egy-két tulajdonságra összpontosítsanak.

Először néhány alapvető komponens, mint gombok, címkék, listapanelek (listbox), szerkesztőmezők, és más ehhez kapcsolódó vezérlőegységek leírása történik. Az ebben a fejezetben tárgyalt komponensek néhánya a Delphi komponenspalettájának Standard lapján, míg mások ettől eltérő lapokon találhatók. A Standard lap elemeit külön szintén azonban nem részletezem; célom minden fejezetben logikailag egymáshoz kapcsolódó komponensek jellemzése, függetlenül a komponenspaletta lapjai által meghatározott sorrendtől.

A Windows saját komponensei

Bizonyára felmerült már a kérdés az olvasóban, hogy miért jó Windows programozáshoz komponenseket használni. A válasz egyszerű: maga a Windows eleve rendelkezik pár komponenssel, ezeket általában vezérlőknek (control) nevezik. Technikailag egy vezérlő nem más, mint egy meghatározott ablak egy speciális működési móddal, tulajdonságokkal és néhány eljárással (bár a hagyományos C nyelvű kód a Windows beépített komponenseit üzenetek küldésével és fogadásával érte el). Ezek jelentették az első lépést a komponensfejlesztés irányában. A következő lépésnek a Visual Basic vezérlőket vehetjük, a harmadik lépést a Delphi komponensei jelentik.

A Windows 3.1 hatféle beépített vezérlőjét általában párbeszédablakokban használják. Ezek gombok (nyomógombok, jelölőnégyzetek és rádiógombok), állandó címkék, szerkesztő mezők, listapanelek, kombinált listák (combo box) és gördítősávok lehetnek. A Windows 95-ben számos új komponens jelent meg, olyanok, mint listatároló (list view), státuszsor, léptetőgomb (spin button), folyamatsáv, fül-vezérlő, és sok más. A programozók már régóta használják ezeket, viszont minden alkalommal, mikor új programot készítettek, újra

Megjegyzés

Valójában a Microsoft harmadik lépése volt az ActiveX vezérlők megjelentetése (a korábbi OCX technológia egy bővítése), melyek a VBX vezérlők utódai. A Delphiben egyaránt használhatunk ActiveX és eredeti komponenseket, de technológia szempontjából a Delphi összetevők sokkal előrébb tartanak az ActiveX vezérlőknél. Elég csak azt nézni, hogy a Delphi komponensek teljes mértékben objektum-orientáltak, míg az ActiveX-eknél az öröklésfogalmat nem implementálták teljesen. Láthatunk ugyan néhány egyszerűbb példát a következő fejezetekben is, az ActiveX vezérlők kezelésének részletezése azonban a 23. fejezetben található.

bele kellett építeniük őket a programjukba. A Windows 95 fejlesztők már használhatják a rendszer által nyújtott standard vezérlőket, a Delphi 3 fejlesztők pedig élvezhetik a könnyen használható komponensek előnyeit.

A standard rendszervezérlők minden Windows alkalmazás alapvető komponensei, tekintet nélkül a használt programozási nyelvre, és minden Windows felhasználó nagyon jól ismeri azokat. A Delphi szó szerint becsomagolja ezeket az előre megadott vezérlőket néhány saját alapvető komponensébe – beleértve az e fejezetben tárgyaltakat is.

Új!

A Delphi 3 a Delphi 2-höz képest tartalmaz néhány általánosabb vezérlőt. Mint később majd látni fogjuk, létezik egy Animate, és egy Toolbar komponens. Ezen kívül megtalálhatók a Windows 95 kiadása utáni Microsoft Windows vezérlők, mint például a CoolBar és a DateTimePicker.

Gombra kattintás

Az első fejezetben kis alkalmazásokat készítettünk, melyek egy gombon alapultak: a gombra való kattintás például egy „Hello" üzenet megjelenését eredményezte a képernyőn. A program másik művelete a gomb mozgatása volt, aminek segítségével az mindig a form közepén jelent meg. Ezután más példákban találkozhattunk olyan nyomógombokkal, melyek egy adott utasítássor végrehajtását szolgálták (OnClick eseményük használatával).

Mostani példánkban egy újabb formot fogunk építeni különböző gombokkal, és menet közben változtatjuk azok tulajdonságait. Egy adott gombon való kattintás egy másik gomb tulajdonságának megváltoztatását fogja eredményezni. A Buttons (gombok) program elkészítéséhez ajánlott először figyelmesen követni az utasításokat, és csak ezután változtatni rajta tetszés szerint. A könyvbéli ismertető elolvasása után természetesen a CD mellékleten lévő forrás állományokon is lehet dolgozni.

A Buttons (gombok) példa

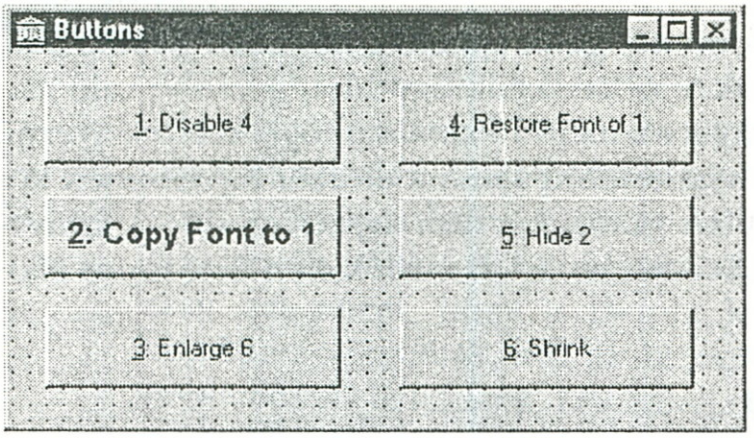
Először nyissunk egy új projektet, majd elmentésekor adjunk neki nevet. Nálam például a formot leíró unit neve BUTTONF.PAS és a projekté BUTTONS.DPR. Ezután készítsünk néhány gombot, mondjuk hatot.

Tipp

Amellett, hogy kiválasztunk egy összetevőt, ráhúzzuk a formra, majd a művelet megismétléséhez újra kiválasztjuk, létezik egy rövidebb út is. Egyszerűen a SHIFT gomb nyomvatartása mellett válasszuk ki az összetevőt a Components palettán való kattintással. A komponens kiválasztva marad, amit kis keret is jelez. Ezután ugyanabból az komponensből több példány is letehető.

Még akkor is ajánlott az **Edit/Align** parancs használata a gombok pontos elrendezéséhez, ha a form rácsához igazítottuk azokat. A hat gomb egyidejű kiválasztása egyaránt történhet egy kijelölő téglalap segítségével, vagy egyenként, a SHIFT billentyű lenyomása mellett. Ez esetben valószínűleg jobb egy három gombból álló oszlopot egyszerre kijelölni, majd a 8.1. ábra szerint elrendezni azokat.

Most, hogy van egy formunk hat gombbal, elkezdhetjük beállítani azok tulajdonságait. Mindenekelőtt a formnak adjunk egy nevet (ButtonsForm) és egy címkét (Buttons). A következő lépésben beállítjuk az egyes gombok Text (szöveg), vagy Caption (címke) tulajdonságait. Egy adott gombhoz tartozó címke általában a felhasználó kattintása által kiváltott esemény leírására szolgál. Követve ezt a szabályt, megszámozzuk a gombokat és az egyes címkék kezdetén ezt a számot írjuk. Így például ha az egyes gomb kikapcsolja a négyest (amely ugyanabban a sorban van), elnevezhetjük 1: Disable 4-nek. A többi gomb címkéjének elkészítése ehhez hasonlóan történhet.



8.1. ábra *A példa elkészítéséhez használt hat elrendezett gomb*

A form komponenseinek tulajdonságai összegezve, szöveges leírásban:

```
object ButtonsForm: TButtonsForm
  Caption = 'Buttons'
  object Button1: TButton
    Caption = '&1: Disable 4'
    OnClick = Button1Click
  end
  object Button2: TButton
    Caption = '&2: Copy Font to 1'
    Font.Color = clBlack
    Font.Height = -15
    Font.Name = 'Arial'
    Font.Style = [fsBold]
    ParentFont = False
    OnClick = Button2Click
  end
  object Button3: TButton
    Caption = '&3: Enlarge 6'
    OnClick = Button3Click
  end
  object Button4: TButton
    Caption = '&4: Restore Font of 1'
    OnClick = Button4Click
  end
  object Button5: TButton
    Caption = '&5: Hide 2'
    OnClick = Button5Click
  end
  object Button6: TButton
    Caption = '&6: Shrink'
    OnClick = Button6Click
  end
end
```

Tipp

Figyeljük meg, hogy minden gombon van egy aláhúzott billentyűparancs, jelen esetben a gomb száma. Egyszerűen elhelyezünk minden gomb címkéjének elejére egy-egy & jelet, mint például '&1: Disable 4', és máris billentyűzetről is elérhető gombokat készítettünk. Csak nyomjunk le egy 7 alatti számot, és az egyik gomb ki lesz jelölve, bár nem látjuk, hogy lenyomódott és felengedett.

A végső lépés természetesen a kívánt működést szolgáltató kód megírása. Mi minden gomb OnClick eseményét kezelni szeretnénk. A legegyszerűbb kód a Button2-é és a Button4-é. A Button2 lenyomásakor a program átmásolja a gomb betűtípusát (ami különbözik a többi gomb standard betűtípusától) a Button1-re, majd hatástalanítja magát:

```
procedure TButtonsForm.Button2Click(Sender: TObject);
begin
  Button1.Font := Button2.Font;
  Button2.Enabled := False;
end;
```

A Button4 lenyomásával visszaáll a gomb eredeti betűtípusa. A font közvetlen másolása helyett a form eredeti betűkészletének visszaállítását a gomb ParentFont tulajdonságának használatával is megtehetjük. Az esemény emellett bekapcsolja a Button2-t, hogy újra használható legyen a Button1 betűtípusának megváltoztatására:

```
procedure TButtonsForm.Button4Click(Sender: TObject);
begin
  Button1.ParentFont := True;
  Button2.Enabled := True;
end;
```

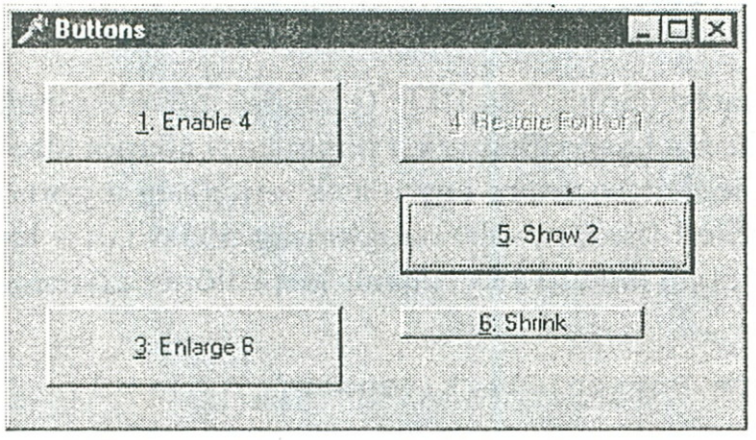
A Button1 és Button5 Disable (hatástalanítás) és Hide (elrejtés) műveleteinek megvalósításához egy logikai változót használhatunk a pillanatnyi állapot tárolására. Az elvégzendő művelet meghatározására egy másik lehetőség a gomb pillanatnyi állapotának – az Enabled (bekapcsolt) tulajdonság állásának – lekérdezése. Amint a következő kódban látható a két eljárás két különböző megközelítést jelent:

```
procedure TButtonsForm.Button1Click(Sender: TObject);
begin
  if not Button4.Enabled then
  begin
```

```
      Button4.Enabled := True;
      Button1.Caption := '&1: Disable 4';
    end
    else
    begin
      Button4.Enabled := False;
      Button1.Caption := '&1: Enable 4';
    end;
end;

procedure TButtonsForm.Button5Click(Sender: TObject);
begin
  Button2.Visible := not Button2.Visible;
  if Button2.Visible then
    Button5.Caption := '&5: Hide 2'
  else
    Button5.Caption := '&5: Show 2';
end;
```

A program futásának eredménye a 8.2. ábrán látható. Az utolsó két gomb programja nem tartalmaz ellenőrzéseket, így például a Button6 annyira összezsugorítható, míg végül teljesen eltűnik:

```
procedure TButtonsForm.Button3Click(Sender: TObject);
begin
  Button6.Height := Button6.Height + 3;
  Button6.Width := Button6.Width + 3;
end;
procedure TButtonsForm.Button6Click(Sender: TObject);
begin
  Button6.Height := Button6.Height - 3;
  Button6.Width := Button6.Width - 3;
end;
```



8.2. ábra *A Buttons példa formjában elrejthetünk vagy kikapcsolhatunk pár gombot, vagy csökkenthetjük valamelyik méretét*

Kattintás az egérrel

Példáinkat mindeddig a gombok OnClick eseményére alapoztuk. Majdnem minden komponens rendelkezik hasonló kattintási eseménnyel, de mit is jelent valójában egy kattintás, és hogyan kapcsolódik más eseményekhez, mint például az OnMouseDown (egérgomb lenyomása) és az OnMouseUp (egérgomb felengedése)?

Először nézzük a kattintást. Első ránézésre az ember azt gondolja, a kattintáshoz nem kell mást tennie a felhasználónak, csak lenyomni, majd elengedni a bal egérgombot a vezérlőn. Ez valóban igaz, viszont a helyzet ennél összetettebb. Amikor a felhasználó a bal egérgombbal egy gomb fölött kattint, az adott komponens képe szintén megváltozik. Ha azonban az egérmutatót a gomb felületén kívülre mozgatja (folyamatosan nyomva tartva a bal egérgombot), az adott gomb képe visszaáll az eredeti (lenyomás előtti) állapotba. Ha a felhasználó ekkor a gomb területén kívül engedi el a bal egérgombot, nem történik semmi, kattintás sem. Másrészről, ha a mutató visszakerül a nyomógombra, az újra benyomódik, és az egérgomb elengedésekor kattintás történik. Ha ez a rész kissé homályos, ki lehet próbálni egy nyomógombon, bármelyik Windows alkalmazás bármelyik gombja megteszi.

Nézzük a kérdés másik felét. A Windows-ban a fent leírt viselkedés a gombokra jellemző, bár a Delphi legtöbb komponensére, például a formokra is kiterjesztette. Emellett a rendszer elemibb eseményeket is generál – egyet az egérgomb lenyomásakor, és egy másikat a gomb felengedésekor. A Delphiben ezeket az eseményeket OnMouseDown-nak és OnMouseUp-nak nevezik. Az egérnek több gombja van, de az egér bármelyik gombjának lenyomásakor ugyanaz az esemény keletkezik.

Szükség lehet azonban arra, hogy megtudjuk, melyik egérgombot nyomta meg a felhasználó. E célból az esemény leírók (event handler) tartalmaznak egy paramétert, amely megadja, melyik gomb lenyomása váltotta ki az eseményt. Az eljárások emellett tartalmaznak még egy további paramétert, amely speciális billentyűk (például SHIFT vagy CTRL) állapotát, és két egész értéket, melyek az esemény helyének x és y koordinátáját adják vissza. Az eseménynek megfelelő eljárás egy formra:

```
procedure TForm1.FormMouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

A legtöbb esetben azonban nincs szükségünk ilyen részletességre, és egyszerűbb a „kattintás" eseményt használni programunkban. A tizedik fejezetben található egy részletesebb példa az egér eseményeinek kezelésére.

Színes szövegek a formon

Most, hogy játszadoztunk egy kicsit a gombokkal, ideje továbblépnünk egy új komponensre, a címkére (Label). A címke tulajdonképpen csak egy formba írt szöveg, vagy megjegyzés. A felhasználó általában egyáltalán nem tesz vele semmit – vagy legalábbis nem közvetlenül. Nincs sok értelme rákattintani egy címkére (bár a Delphiben ez technikailag megoldható). Megjegyzendő az is, hogy a formban látható szövegek nem mindegyike kapcsolódik feltétlenül címkéhez, mivel az (mint minden komponens) egyszerűen írhat bármit saját felületére, például a TextOut eljárással.

A címkéket más komponensek jellemzésére használjuk, különösen szerkesztő mezők, listák, vagy kombinált listák (combo box) használatakor, mivel ezek nem rendelkeznek saját felirattal. Ha egy tetszőleges Windows alkalmazásban megnyitunk egy párbeszédablakot, bizonyára találkozunk feliratokkal. Ezek statikus vezérlők (Windows terminológia), azaz címkék (Delphi terminológia).

Új!

A Windows a címkéket statikus osztályú ablakokként implementálja, a Delphi ehelyett grafikus komponensként. Ez nagyon fontos, mivel ez a módszer a form elkészítését is elősegíti, és még némi Windows erőforrást is megtakarít. A Delphi 3 azonban tartalmaz egy új összetevőt, ami a Windows címkének felel meg, ez pedig a StaticText (statikus szöveg) komponens. Hasonló tulajdonságokkal és ugyanazokkal az eseményekkel rendelkezik, a Borland valószínűleg az ActiveX támogatás miatt készítette. Következő példánkban használni fogjuk ezt az komponenst.

A leírások mellet alkalmazásunk felületének kicsinosítására is használhatjuk ezt a komponenst, mint azt következő példánkban, a LabelCo-ban is megfigyelhetjük. Az alkalmazást azért készítettem, hogy megmutassam, hogyan lehet futási időben megváltoztatni a címke tulajdonságait (konkrétan a program a szöveg hátterének és betűinek színét, valamint a szöveg igazítását befolyásolja).

A LabelCo példa

Az első teendő egy nagy címke elhelyezése a formban és némi szöveg begépelése, nem árt valami hosszabbat írni. Ajánlatos a WordWrap (szövegtördelés) tulajdonságot True-ra (igaz) állítani az esetleges többsoros szöveg miatt, és az

AutoSize-t (automatikus méret) pedig False-ra (hamis), hogy a címke szabadon méretezhető legyen. Ezen kívül jó, ha egy nagyobb méretű betűtípust, és jól látható színeket választunk.

A betűkészlet-, ill. háttérszín, és az igazítás tulajdonságok menet közbeni változtatására gombokat használunk. Ahelyett, hogy ezeket a gombokat közvetlenül a formra tennénk, inkább elhelyezünk egy panelt a formon, és azon helyezzük el a szükséges öt komponenst. A színek változtatására kettő, míg az igazítás kiválasztására további három gombra van szükség – balra, középre, ill. jobbra igazítás. A gombokat a panelra tettük, így az lesz az egyes gombok Parent (szülő) komponense: a koordináták a panelhoz viszonyítottak, így annak mozgatása a gombokat is mozdítja, elrejtése pedig elrejti a gombokat is, és így tovább.

Megjegyzés

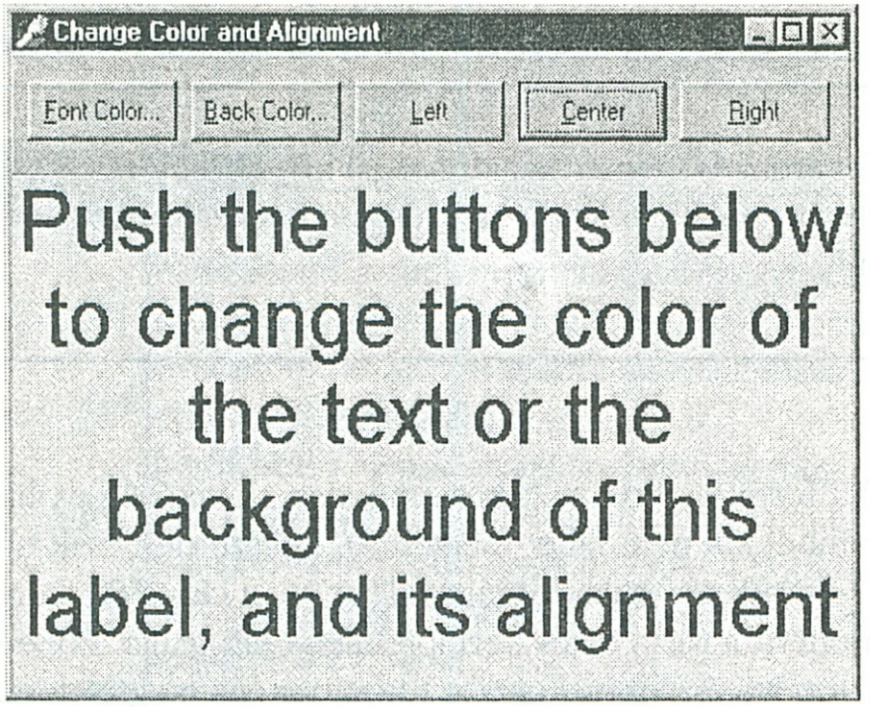
Egy komponens Parent és Owner tulajdonságai közötti különbségek a legutóbbi fejezetben találhatók. Ha ez a rész nem teljesen tiszta, ott utána lehet nézni.

Mivel a címke és a panel a form leszármazott komponensei, lehetőség nyílik az elrendezésükre (ezt gombokkal nem lehet megtenni). Egyszerűen csak be kell állítani a panel Align (elrendezés) tulajdonságát alTop-ra, a címkéét pedig alClient-re, és a két komponens a form teljes kezelőfelületét elfoglalja. Ebben az a különleges, hogy a form átméretezésekor a két komponens megfelelően módosítja méretét és pozícióját. Vegyük észre, hogy címkével ellentétben a StaticText komponens nem rendelkezik Align tulajdonsággal.

Az utolsó komponens, melyet a formra kell helyeznünk a ColorDialog (A Components paletta Dialogs oldalán található), mely a standard Windows színválasztó párbeszédablakot nyitja meg. Az imént összerakott formot a 8.3. ábra mutatja, listája pedig az alábbi:

```
object ColorTextForm: TColorTextForm
  Caption = 'Change Color and Alignment'
  object Label1: TLabel
    Align = alClient
    Alignment = taCenter
    Caption = 'Push the buttons...' // omitted
    Color = clYellow
    Font.Color = clNavy
    Font.Height = -40
```

```
      Font.Name = 'Arial'
      WordWrap = True
    end
    object Panel1: TPanel
      Align = alTop
      object BtnFontColor: TButton...
      object BtnBackColor: TButton...
      object BtnLeft: TButton...
      object BtnCenter: TButton...
      object BtnRight: TButton...
    end
    object ColorDialog1: TColorDialog...
end
```



8.3. ábra *A színes címke példájának formja futás közben*

Megjegyzés

A szín megváltoztatására használt két gomb egy párbeszédablakot jelenít meg, nem cselekszik azonnal. Ebből kifolyólag címkéjük végén három pont található, amely egy standard Windows konvenció a párbeszédablakot nyitó menüpontok és gombok jelölésére.

Most már itt az ideje, hogy programozzunk is egy kicsit. A három igazításra használt gomb eljárásai nagyon egyszerűek. A programnak a címke igazítását kell állítania:

```
procedure TColorTextForm.BtnLeftClick(Sender: TObject);
begin
  Label1.Alignment := taLeftJustify;
end;
```

A másik két eljárásnak a taLeftJustify helyett a taCenter és taRightJustify értékeket kell használnia. Az Object Inspector segítségével a címke Alignment tulajdonságának három lehetséges értéke megtekinthető.

A színt megváltoztató kód megírása kicsit bonyolultabb. Tulajdonképpen egy új színértéket kell szolgáltatnunk, esetleg egy listából választva ki néhány lehetséges érték közül. Megoldhatnánk a problémát egy színekből álló tömb deklarálásával, néhány érték bevitelével, és a gombokkal előre- és visszalépkedhetnénk a tömb elemei között, de létezik egy elegánsabb megoldás, amely kevesebb kódot is igényel: a Windows standard párbeszédablakának használata a szín kiválasztására.

A standard szín párbeszédablak

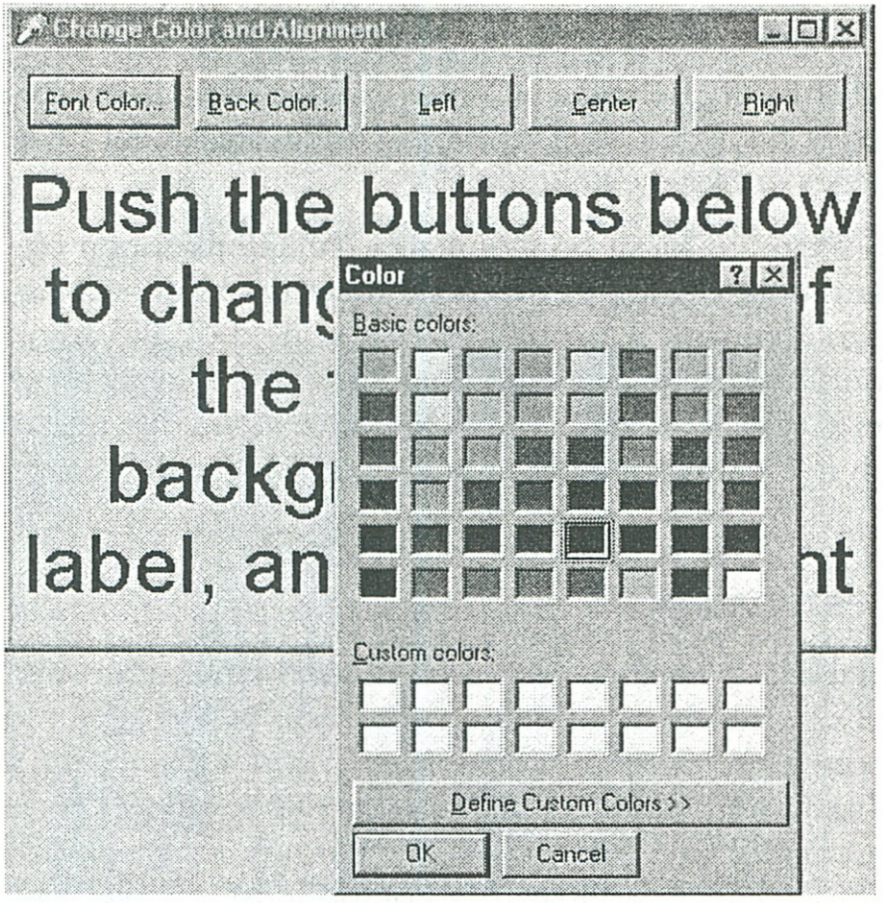
A standard szín párbeszédablak használatához a Delphi komponenspalettájának Dialogs oldalán lévő ColorDialog komponenst válasszuk ki, és tegyük le bárhol a formon. A komponens pozíciója nem lényeges, mivel a formon ez a komponens nem látszik (az itt leírt standard párbeszédablakok egy részletesebb bemutatása a tizenharmadik fejezetben található, bár sok azt megelőző fejezetben is használjuk). Most már használhatjuk a komponenst, a következő kód megírásával:

```
procedure TColorTextForm.BtnFontColorClick(Sender:
TObject);
begin
  ColorDialog1.Color := Label1.Font.Color;
  if ColorDialog1.Execute then
    Label1.Font.Color := ColorDialog1.Color;
end;
```

Az eljárás törzsében szereplő három sor jelentése a következő: az elsőben a párbeszédablakban jelzett kezdeti színnek a címke háttérszínét választjuk ki; a másodikban elindítjuk a párbeszédablakot; a harmadikban pedig a felhasználó által kiválasztott színt beírjuk a címkébe. Ez utóbbit csak akkor tesszük, ha a párbeszédablak bezárása az **OK** gombbal történt (ez esetben az Execute eljárás igaz értéket ad). A **Cancel** gomb lenyomásakor (ekkor az Execute hamis értékkel tér vissza) átlépjük az utolsó utasítást. A címke szövegszínének megváltoztatásához hasonló kódot írunk, csak ennél a Label1.Font.Color

tulajdonságra hivatkozunk (a LabelCo példa teljes forráskódja a CD-n a LABELF.PAS állományban található).

A szín párbeszédablak, mely a Define Custom Colors (Egyéni színek definiálása) gombra kattintva kiszélesíthető, a 8.4. ábrán működés közben látható.



8.4. ábra *A címke színének és szövegének kiválasztására használt párbeszédablak (a felhasználó most éppen a szöveg színét állítja)*

Elemek áthúzása komponensek között

Más Delphi komponensek kipróbálása előtt sokat segíthet egy különleges technika megvizsgálása: az áthúzás. A húzás művelet meglehetősen egyszerű, és egyre inkább általános a Windowsban. Windows 95-ben állományok áthúzhatók egyik mappából a másikba, az Asztalra (Desktop) tehetők, vagy a Windows Intéző segítségével különböző fájl- és mappaműveletek végezhetők ilyen módon. Maga a folyamat általában az egérgombnak az egyik komponensen (ill. ablakon) való lenyomásából, és a másik komponensen (ill. ablakon) való felengedéséből áll. A művelet után végrehajtható a kód, ez általában egy tulajdonság, egy érték, vagy valami hasonló bemásolása a célkomponensbe.

Példaként tekintsük a 8.5. ábrán látható formot, ahol négy színes címke található, mindegyiken az egyes színek nevével, valamint egy célcímke, néhány sornyi szöveggel. A célcímke valójában egy StaticText komponens, melynek Border (keret) tulajdonságának speciális értéke van, az sbsSunken, ami besüllyesztett hatást kelt. A hagyományos címkék nem rendelkeznek ilyen lehetőségekkel. A Dragging nevű példa célja, hogy a statikus szöveg bal oldalán álló címkékről ki lehessen választani egy színt, és a szövegre ráhúzva annak háttere megfelelően változzon. A komponenseknek nagyon egyszerű tulajdonságaik vannak, mint ahogyan az alábbi szöveges leírás is mutatja:

```
object DraggingForm: TDraggingForm
  Caption = 'Dragging'
  Font.Color = clBlack
  Font.Height = -16
  Font.Name = 'Arial'
  Font.Style = [fsBold]
  object LabelRed: TLabel
    Alignment = taCenter
    AutoSize = False
    Caption = 'Red'
    Color = clRed
    DragMode = dmAutomatic
  end
  object LabelAqua: TLabel...
  object LabelGreen: TLabel...
  object LabelYellow: TLabel...
  object StaticTarget: TStaticText
    Alignment = taCenter
    AutoSize = False
    BorderStyle = sbsSunken
    Caption = 'Drag colors here to change the color'
    Font.Height = -32
    OnDragDrop = StaticTargetDragDrop
    OnDragOver = StaticTargetDragOver
  end
end
```

A címkék előkészítése után, mely azok nevének és címének, valamint egy megfelelő szín megadásával történik, a húzást engedélyezni kell. Ez megtehető a négy bal oldali címke DragMode tulajdonságának dmAutomatic-ra való állításával, és a célcímkénél néhány esemény kezelésével.

Megjegyzés

Az automatikus húzás módhoz kapcsolódóan kiválaszthatjuk a kézi áthúzás módot is. Ez a BeginDrag és EndDrag eljárások használatán alapul. Ezt a technikát a 11. fejezet egyik példájában láthatjuk.

8.5. ábra *A Dragging példa formja*

A Dragging példa kódja

Az első esemény, mellyel foglalkoznunk kell, az OnDragOver, mely az egérmutatónak az adott komponensre való húzásakor hívódik meg. Ez az esemény jelzi, hogy a komponens elfogadja-e azt, hogy ráhúznak valamit. Legtöbbször ez annak eldöntése után történik, hogy a forráskomponens (ahonnan a művelet ered) egy adott típusú-e:

```
procedure TDraggingForm.StaticTargetDragOver(
  Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept := Source is TLabel;
end;
```

Ez a kód elfogadja a műveletet, aktiválja megfelelő egérkurzort, feltéve, hogy a forrás objektum valóban egy Label komponens. Itt megfigyelhető az „is" dinamikus típusellenőrző operátor használata.

A második megírandó eljárás az OnDragDrop eseményhez kapcsolódik:

```
procedure TDraggingForm.StaticTargetDragDrop(
          Sender, Source: TObject;  X, Y: Integer);
begin
 StaticTarget.Color := (Source as TLabel).Color;
end;
```

A forrásobjektum Color (szín) jellemzőjének kiolvasásához át kell alakítanunk az objektumot a megfelelő adattípusra, jelen esetben TLabel-re. Egy típuskonverziót kell tehát végrehajtanunk – szaknyelven szólva egy downcast-et

(típusátalakítás egy adott osztályból annak egy leszármazott osztályába, a hierarchia szerint lefelé). Mint az ötödik fejezetben láthattuk, egy ilyen típus átalakítás nem biztonságos. Ennek az átalakításnak tulajdonképpen az a lényege, hogy a Source (forrás) paramétert, ami valójában egy TLabel, mi TObject típusban kapjuk, viszont szeretnénk úgy használni, mint egy valódi TLabel objektumot, ami a TObject leszármazottja. Általánosságban azonban szembe kell néznünk egy olyan objektum TLabel oszttályúvá alakításának veszélyeivel, mely eredetileg nem címke, hanem mondjuk egy gomb. Egy gomb címkeként való használata futás közbeni hibákat eredményezhet.

Az „as" operátor minden használatakor típusellenőrzés történik. Ha a forrás objektum nem TLabel, a Delphi kivételt vált ki. Ebben az esetben azonban ezzel nem kell törődnünk, mivel az OnDragDrop eseményhez csak az OnDragOver eljárás által beállított igaz Accept érték esetén jutunk, amit mi csak TLabel típusú forrás objektum esetén engedünk meg.

Felhasználói input fogadása

Számos módját láttuk annak, ahogyan a felhasználó az egér segítségével az általunk írt alkalmazással kommunikál: egérkattintások, áthúzások, és így tovább. És a billentyűzet? Mint tudjuk, az egér használata helyett a billentyűzet segítségével is kiválaszthatunk egy adott gombot a rajta lévő címke aláhúzott karakterének lenyomásával.

Néhány különleges esettől eltekintve a Windows a billentyűzetet közvetlenül kezeli. A billentyűzetfüggő események részére kialakított kezelők egyébként nem gyakoriak. A rendszer valójában kész vezérlőket szolgáltat szerkesztő mezők, és egy egyszerű szövegszerkesztő beépítésére. A Delphi több különböző komponenssel rendelkezik ezen a téren. Edit (szerkesztőmező), MaskEdit (maszkolt szerkesztőmező), Memo (feljegyzés), RichText (formázott szöveg), és az ezekhez tartozó adatfigyelő (data-aware) vezérlők. A két alapvető komponens az Edit és a Memo.

A szerkesztőmező mindössze egyetlen sor szöveg szerkesztését teszi lehetővé, olyan sajátos tulajdonságokkal, mint például a karakterek maximális számának korlátozása, vagy jelszók lekérdezéséhez speciális karakter megjelenítése a gépelt szöveg helyett. A Memo komponens, mint hamarosan kiderül, több sor tárolását teszi lehetővé.

A szerkesztőmezőkre vonatkozó, Focus nevű első példában egy sok vezérlőben megjelenő elemet, az bemeneti fókuszt (input focus) mutatjuk be. A Windowsban meglehetősen könnyű megállapítani, hogy melyik az aktuális fő ablak: az, amelyik minden más ablakot eltakar, és a címsora megkülönböztetett színű. Nem ilyen egyszerű azonban annak megállapítása, hogy melyik ablak (vagy komponens) rendelkezik a bemeneti fókusszal. Ha a felhasználó lenyom egy gombot, melyik komponens kapja meg a megfelelő üzenetet a

billentyűzetről? Lehet, hogy az aktív ablak, de lehet, hogy vezérlőinek valamelyike. Nézzünk egy formot néhány szerkesztőmezővel. Egyszerre csak az egyiknél lehet a fókusz, viszont a felhasználó továbbadhatja azt a tabulátor-billentyű segítségével, vagy egy másik komponensen való kattintással.

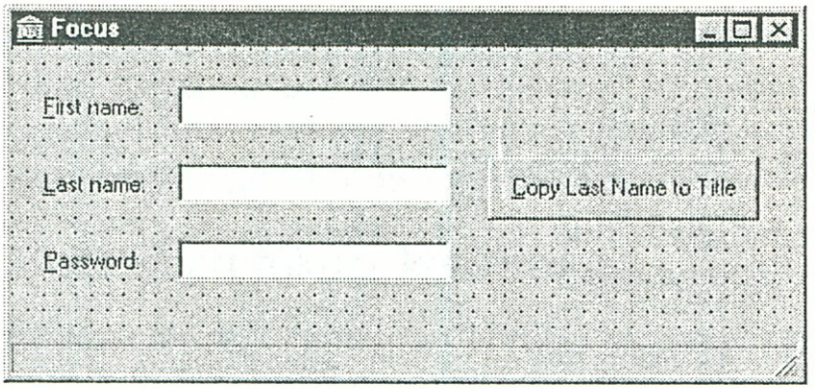
A bemeneti fókusz kezelése

Példánk lényege (amely a CD-n Focus címmel található meg), hogy valahányszor egy komponens megkapja, vagy elveszíti a fókuszt, egy megfelelő eseményt kap, amely jelzi, hogy a felhasználó elérte (OnEnter), vagy elhagyta (OnLeft) az adott komponenst. Ennek segítségével kiegészíthetjük a formot a fókuszt kezelő eljárásokkal, és a megfelelő információkat jelezhetjük egy címkében, vagy a státuszsorban.

A három szerkesztő mező mellett a formon címkék is szerepelnek, melyek az egyes szerkesztő mezők jelentését tartalmazzák (First name (keresztnév), Last name (vezetéknév), és Password (jelszó)). A példához használt form a 8.6. ábrán található. A pillanatnyi állapot kijelzésére a programban egy speciális Win95 komponens, a StatusBar (státuszsor) szerepel, de egy címke, vagy egy panel használata is hasonló eredményt szolgáltat. Valójában a StatusBar komponenst SimplePanel tulajdonságának True-ra való állításával egy egysoros kimeneti eszközként lehet használni. Íme a példában szereplő tulajdonságok:

```
object FocusForm: TFocusForm
  Caption = 'Focus'
  object Label1: TLabel
    Caption = '&First name:'
    FocusControl = EditFirstName
  end
  object Label2: TLabel
    Caption = '&Last name:'
    FocusControl = EditLastName
  end
  object Label3: TLabel
    Caption = '&Password:'
    FocusControl = EditPassword
  end
  object EditFirstName: TEdit
    TabOrder = 0
    OnEnter = EditFirstNameEnter
  end
  object EditLastName: TEdit
    TabOrder = 1
    OnEnter = EditLastNameEnter
  end
  object EditPassword: TEdit
    PasswordChar = '*'
    TabOrder = 2
```

```
    OnEnter = EditPasswordEnter
  end
  object ButtonCopy: TButton
    Caption = '&Copy Last Name to Title'
    TabOrder = 3
    OnClick = ButtonCopyClick
    OnEnter = ButtonCopyEnter
  end
  object StatusBar1: TStatusBar
    SimplePanel = True
  end
end
object FocusForm: TFocusForm
  Caption = 'Focus'
  object Label1: TLabel
    Caption = '&First name:'
    FocusControl = EditFirstName
  end
  object Label2: TLabel
    Caption = '&Last name:'
    FocusControl = EditLastName
  end
  object Label3: TLabel
    Caption = '&Password:'
    FocusControl = EditPassword
  end
  object EditFirstName: TEdit
    TabOrder = 0
    OnEnter = EditFirstNameEnter
  end
  object EditLastName: TEdit
    TabOrder = 1
    OnEnter = EditLastNameEnter
  end
  object EditPassword: TEdit
    PasswordChar = '*'
    TabOrder = 2
    OnEnter = EditPasswordEnter
  end
  object ButtonCopy: TButton
    Caption = '&Copy Last Name to Title'
    TabOrder = 3
    OnClick = ButtonCopyClick
    OnEnter = ButtonCopyEnter
  end
  object StatusBar1: TStatusBar
    SimplePanel = True
  end
end
```



8.6. ábra *A Focus példa formja*

Mint látható, a form még egy gombot is tartalmaz, mellyel a form címkéjét a LastNameEdit-ben szereplő szöveggel helyettesíthetjük. Ez csak egy példa arra, hogy milyen módon lehet egy szerkesztőbe írt szöveggel dolgozni. A következő kódból is látszik, hogy egy szerkesztőmező Text (szöveg) tulajdonságának használata előtt ajánlatos megvizsgálni, hogy a felhasználó begépelt-e egyáltalán valamit, vagy a mező üres:

```
procedure TFocusForm.ButtonCopyClick(Sender: TObject);
begin
  if EditLastName.Text <> '' then
    FocusForm.Caption := EditLastName.Text;
end;
```

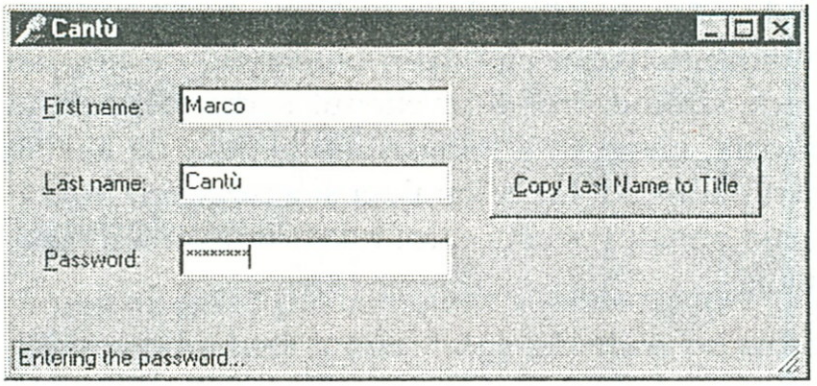
Most már rátérhetünk a program legérdekesebb részére. Amint a 8.7. ábrán is látható, valahányszor a fókusz egy másik vezérlőre kerül, egy megjegyzést írhatunk a form alján lévő státuszsorba.

Tipp

A felhasználónak az alkalmazás lépéseiről való tájékoztatását megoldhatjuk a státuszsorba írt szöveggel, ami a fókusz egyik vezérlőről a másikra lépésekor megfelelően módosul.

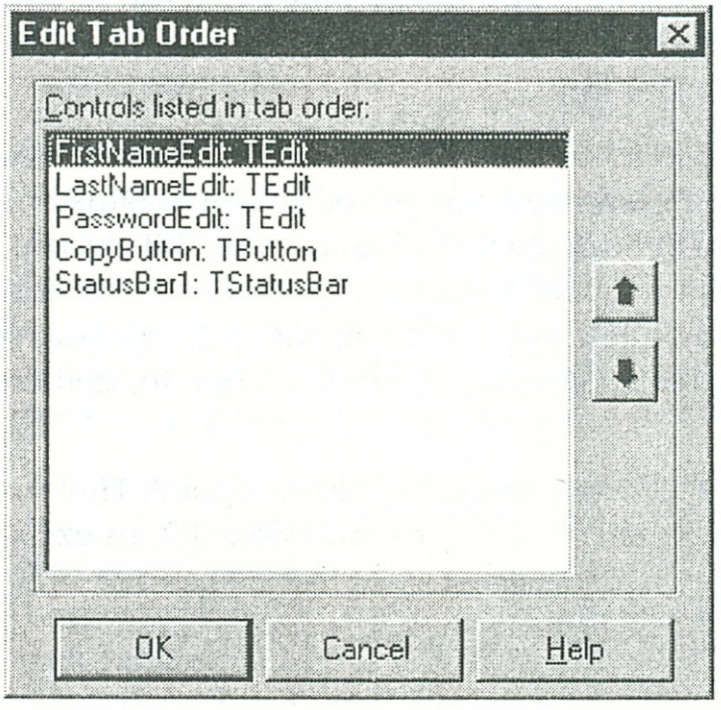
Ennek megvalósításához négy eljárás szükséges, egy minden egyes Edit komponenshez és még egy a gombra, az OnEnter eseményekhez kötődően. Íme az egyik (a másik három nagyon hasonló):

```
procedure TFocusForm.EditFirstNameEnter(Sender:TObject);
begin
  StatusBar1.SimpleText :='Entering the first name...';
end;
```



8.7. ábra *A státuszsorban közölt üzenetek egyike a Focus program futása közben. Figyeljük meg a form új címkéjét*

A programot az egérrel és a billentyűzettel egyaránt letesztelhetjük. A tabulátor lenyomásakor a bemeneti fókusz sorban lépeget a szerkesztő komponenseken és a nyomógombon, a címkék figyelembe vétele nélkül. Az ablakban szereplő komponensek TabOrder tulajdonságának megfelelő beállításával a sorrend módosítható. Ez megtehető a kívánt érték beírásával az Object Inspectorban, sokkal jobb és könnyebb azonban a form gyorsmenüjéből a **Tab Order** parancscsal elérhető Edit Tab Order (tabulárot sorrend szerkesztése) párbeszédablak használata. A 8.8. ábrán a Focus példához tartozó kép szerepel. Látható, hogy a státuszsor szerepel ugyan, viszont a TAB gomb segítségével nem tudunk ráállni.



8.8. ábra *Az Edit fül Order párbeszédablaka a Focus példa fő formjához*

Egy komponens kiválasztásának másik módja a billentyűparancs használata. Nyomógombra könnyen el tudunk helyezni egy ilyen billentyűt, de hogyan ugorhatunk közvetlenül a szerkesztőmezőre? Közvetlenül semmiképpen (a szerkesztő doboz szövege változik, ahogyan a felhasználó gépel), de létezik egy indirekt módszer. A billentyűparancsot (az & karakter segítségével) hozzárendelhetjük egy címkéhez, majd annak FocusControl tulajdonságát a megfelelő Edit komponensre állíthatjuk.

Tipp

A Windows 95-ben a szerkesztő vezérlők automatikusan tartalmaznak a jobb egérgomb megnyomására előugró helyi menüket. Bár egy ilyen menüt könnyű módosítani a Delphiben (mint azt a következő fejezetben láthatjuk), fontos belátnunk, hogy ez egy természetes dolog a Windows 95-ben.

Egy általános OnEnter eseménykezelő

Az előző kód problémája, hogy négy különböző OnEnter eseménykezelőt kell írnunk, melyek a StatusBar komponens szövegébe másolnak egy-egy karakterláncot. Ha a példát további szerkesztőmezőkkel szeretnénk bővíteni, akkor újabb eseménykezelőket kellene definiálni a kód további másolgatásával. Ahhoz, hogy a megjelenítést kissé megváltoztassuk (például hogy több panelt is engedélyezzünk) az összes kódot át kell írni.

A megoldás egy önálló eseménykezelő írása, mely minden szerkesztőmező (és a gomb) OnEnter eseményét kezeli. Egyszerűen csak el kell tárolni a státuszsor üzenetét egy tulajdonságban, majd a Sender objektumban erre kell hivatkozni. A Hint (tipp) tulajdonság erre a célra nagyon megfelelő, igazából arra tervezték, hogy információkat szolgáltasson a felhasználónak az adott elemre vonatkozóan (a Hint tulajdonság részletes tárgyalása a tizenkettedik fejezetben található).

Egyszerűen eltároljuk a kívánt üzenetet a szerkesztőmezők, és a gomb Hint tulajdonságában, majd eltávolítjuk az aktuális OnEnter eseménykezelőt, és ezt írjuk be mindegyik helyett:

```
procedure TFocusForm.GlobalEnter(Sender: TObject);
begin
  StatusBar1.SimpleText := (Sender as TControl).Hint;
end;
```

Vegyük észre, hogy nem írhatunk „Sender as TEdit"-et, mivel a vezérlő akár egy gomb is lehet. Ennek megoldására típuskonverziót használunk a TButton és a TEdit egy közös ős osztályába, mely a Hint tulajdonságot tartalmazza, ahogy az a fenti kódban is látható.

Számok bevitele

Amint az előző példa mutatja a felhasználót egy szerkesztőmező használatával könnyen kérhetjük arra, hogy gépeljen be szöveget, mégha az egy sorra korlátozódik is. Általában azonban sokkal gyakoribb egy programban a numerikus adatok bevitele. Ennek megvalósításához használhatjuk a MaskEdit (maszkolt szerkesztőmező) komponenst (a Components paletta Additional oldalán), vagy egyszerűen az Edit komponens szövegét alakítjuk egésszé a hagyományos Pascal Val függvény, vagy a Delphi IntToStr függvényének használatával.

Ez jól hangzik, de mi történik, ha a felhasználó betűt ír, holott számot várunk? Természetesen az átalakító függvények hibakóddal térnek vissza, melyekkel ellenőrizhetjük, hogy tényleg számot kaptunk-e. A második kérdés, hogy mikor hajtjuk végre ezt az ellenőrzést? Egyaránt lehetséges a szerkesztőmező értékének módosításakor, a fókusz elvesztésekor, vagy amikor a felhasználó egy adott gombra kattint, például egy párbeszédablak OK gombjára. Mint látni fogjuk, ezen technikák nem mindegyike helytálló.

Van egy másik, merőben különböző megoldás, mely csak számok bevitelét engedélyezi. Figyelni kell a szerkesztő dobozba érkező inputot, és visszautasítani a nem numerikus karaktereket. Ez a technika nem teljesen biztonságos (mivel a vágólapról bármilyen szöveget beilleszthetünk a mezőbe), de jól működik és könnyű megvalósítani. Természetesen továbbfejleszthető más technikákkal való kombinálásával.

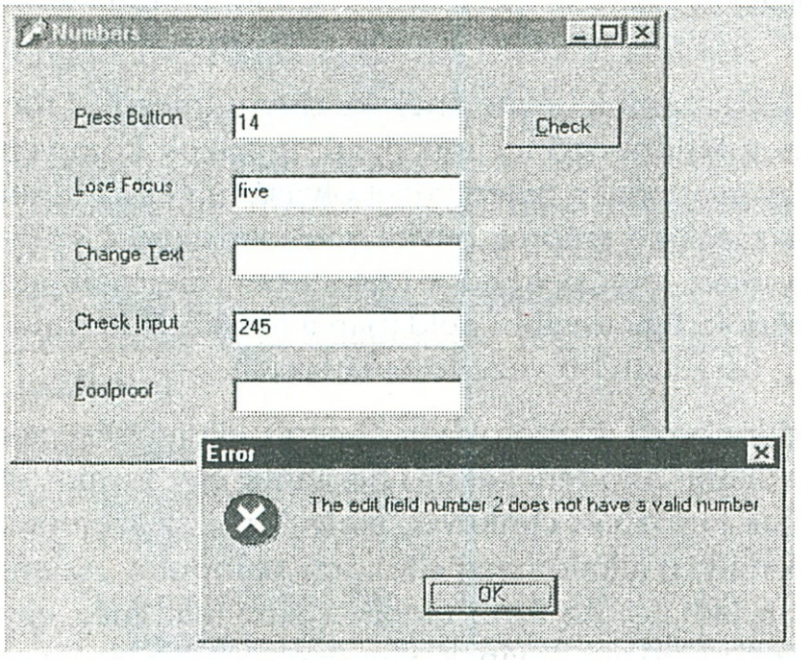
A következő, Numbers nevű példa számok bevitelének szerkesztőmezővel történő megvalósítását mutatja meg különböző technikákkal, így azok könnyen összehasonlíthatók. A példa egy gyakorlat a billentyűzet olvasására és az input fókusz kezelésére. Egy alkalmazásban a számok bevitelének kezelésére szükség lehet speciális komponensekre, mint például a Delphi-ben megtalálható SpinEdit vagy az UpDown vezérlők. Egy további példában a MaskEdit komponensvel a billentyűzet egy még összetettebb vezérlését láthatjuk.

Ebben a példában az input különböző állapotokban történő ellenőrzésének hatásait szeretnénk összehasonlítani. Mindenekelőtt felépítünk egy formot öt szerkesztőmezővel és a hozzájuk tartozó öt címkével, melyekben az szerepel, hogy az adott Edit komponens milyen alkalommal ellenőrzi az inputot. Az ennek megfelelő kép a 8.9. ábrán látható. Az form emellett tartalmaz még egy nyomógombot az első szerkesztő mező tartalmának ellenőrzésére. Az első mezőnél az ellenőrzés a nyomógomb lenyomásakor történik. A gomb OnClick

eseményének leírásában a szöveget először számmá alakítjuk a Val eljárás segítségével, mely egy hibakóddal tér vissza, majd az értéktől függően a felhasználó kap egy üzenetet:

```
procedure TNumbersForm.CheckButtonClick(Sender:TObject);
var
  Number, Code: Integer;
begin
  if Edit1.Text <> '' then
  begin
    Val (Edit1.Text, Number, Code);
    if Code <> 0 then
    begin
      Edit1.SetFocus;
      MessageDlg ('Not a number in the first edit',
        mtError, [mbOK], 0);
    end
    else
      MessageDlg ('OK, the number in the first edit
        box is'+
      IntToStr (Number), mtInformation, [mbOK], 0);
  end;
end;
```

Hiba esetén az alkalmazás még a hibaüzenet megjelenítése előtt visszaállítja a fókuszt a szerkesztő mezőre, felkérve a felhasználót a helyes érték beírására. Ebben a példaalkalmazásban természetesen a felhasználó figyelmen kívül hagyhatja ezt, és átállhat egy másik mezőre.



8.9. ábra *A kijelzett hibaüzenet, amikor a második szerkesztő doboz elveszti a fókuszt és a felhasználó betűket írt*

A fókusz elvesztésekor a második mezőnél hasonló ellenőrzés történik. Ebben az esetben az üzenet automatikusan megjelenik, de csak akkor, ha hibás az input (lásd 8.9. ábra). Miért is zaklatnánk a felhasználót, ha minden renden van? Az itteni kód különbözik az elsőnél lévőtől; nem hivatkozik az Edit2 komponensre, hanem mindig a Sender vezérlőt használja, hogy biztonságos legyen a típusváltás. Az egyes mezők számának meghatározására a Tag tulajdonságot használjuk, melybe a szerkesztőmező vezérlő sorszámát kell beírni.

Tipp

Mint a következő listában látható, a Sender paraméter ugyanabban az eljárásban több alkalommal történő TEdit osztályba alakítása helyett jobb a műveletet egyszer elvégezni, az érték egy TEdit típusú helyi változóban való tárolásával. Az ilyen átalakítások közben fellépő ellenőrzések igazából meglehetősen lassúak.

Az alábbi eljárás valamivel összetettebb, de egy későbbi komponensnél újra használni tudjuk:

```
procedure TNumbersForm.Edit2Exit(Sender: TObject);
var
  Number, Code: Integer;
  CurrEdit: TEdit;
begin
  CurrEdit := Sender as TEdit;
  if CurrEdit.Text <> '' then
  begin
    Val (CurrEdit.Text, Number, Code);
    if Code <> 0 then
    begin
      CurrEdit.SetFocus;
      MessageDlg ('The edit field number ' +
        IntToStr (CurrEdit.Tag) + ' does not have a
            valid number',
        mtError, [mbOK], 0);
    end;
  end;
end;
```

Tipp

A Delphi 32-bites verziójában ez a kód egy figyelmeztető üzenetet eredményez fordítás közben, mert a Number változót nem használjuk miután értéket adunk neki. A figyelmeztető üzenet elhárítása érdekében a $HINTS fordító direktívával belülről az adott eljárásra kikapcsolhatjuk a figyelmeztető üzenetet. Egyszerűen csak be kell írni az eljárás elé, hogy {$HINTS OFF}, utána pedig, hogy {$HINTS ON}, mint ahogyan a példa forráskódjában is szerepel.

A harmadik Edit komponens hasonló ellenőrzést hajt végre, valahányszor változik a tartalma (az OnChange eseményt használjuk). Bár az inputot különböző alkalmakkor ellenőrzi – más eseményeket használva – a három függvény nagyon hasonló. Az alapötlet a már begépelt karakterlánc ellenőrzése.

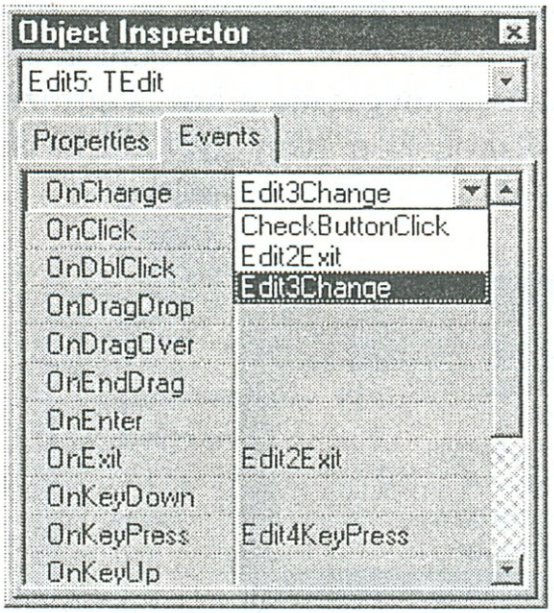
A negyedik Edit komponensnél egy teljesen más technika szerepel. Az ellenőrzést még azelőtt szeretnénk végrehajtani, mielőtt az Edit esemény tudomást szerezne arról, hogy valamelyik gombot lenyomták. Az Edit komponens rendelkezik egy OnKeyPress eseménnyel, mely a felhasználó tevékenységével függ össze. Írhatunk egy eljárást erre az eseményre, és ellenőrizhetjük, hogy a karakter egy szám-e, vagy esetleg a Backspace billentyű (melynek számértéke 8, így a karakterre #8-ként hivatkozhatunk). Ha egyik sem, a karakter értékét „null" karakterre (#0) módosítjuk, amit a szerkesztő nem vesz figyelembe, de ad egy apró figyelmeztető hangot:

```
procedure TNumbersForm.Edit4KeyPress(
  Sender: TObject; var Key: Char);
begin
  {check if the key is a number or backspace}
  if not (Key in ['0'..'9', #8]) then
  begin
    Key := #0;
    Beep;
  end;
end;
```

A negyedik Edit komponens csak számokat fogad el a bementről, de nem teljesen üzembiztos. A felhasználó szöveget másolhat a Vágólapra és beillesztheti azt ebbe az Edit vezérlőbe a SHIFT+INS billentyűkombinációval (nem a CTRL+V-vel), átlépve minden előző ellenőrzést. A probléma elhárítása érdekében második komponensként gondolhatunk még a tartalom változtatásakor történő ellenőrzésre a harmadik mező alapján, vagy az elhagyáskor nézzük meg ugyanezt, mint a negyedik mezőnél. Az előzőek eredményezték az ötödik, a

„Foolproof" feliratú szerkesztőmezőt: használja a negyedik OnKeyPress eseményét, a harmadik OnChange eljárását, és az OnExit-et a másodikból, így nem igényel további kódot.

Egy létező eljárás új eseményben való használatához válasszuk az Object Inspector Events oldalát, álljunk rá az adott komponensre, és az esemény nevén való dupla kattintás helyett válasszuk ki a jobb oldali combo box-ban lévő gombot. Mint az a 8.10. ábrán látható, ekkor az aktuális eseménnyel kompatibilis – ugyanannyi paraméterrel rendelkező – régi eljárások listája jelenik meg.



8.10. ábra *Az Edit5 összetevő más komponensekre írt eljárást használ. Az Object Inspector kombinált listájának kijelölésekor minden kompatibilis eljárás megjelenik*

A megfelelő eljárások kiválasztása után az ötödik komponens ötvözni fogja a harmadik és a negyedik tulajdonságait. Ez azért lehetséges, mert az iménti eljárások sehol sem hivatkoznak a hozzájuk tartozó vezérlőre.

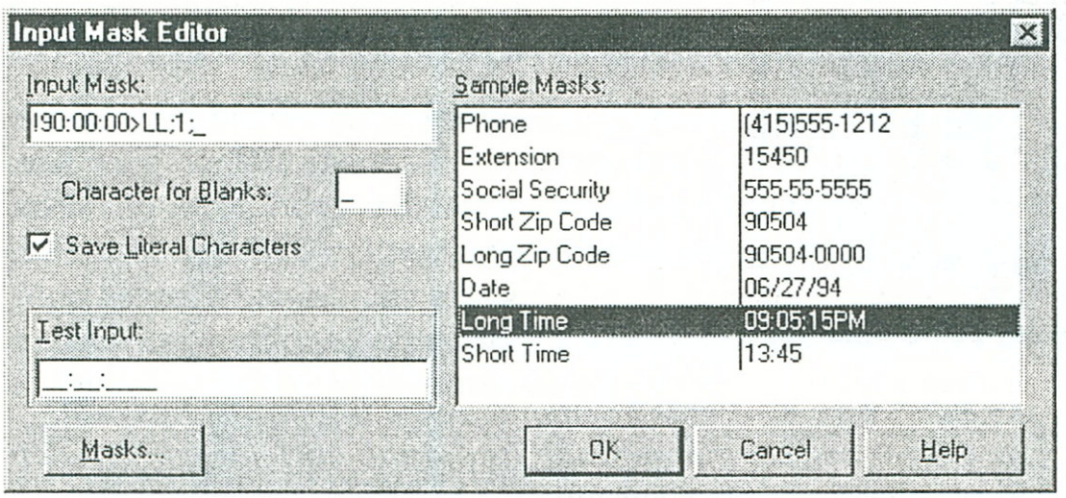
A használt technika az általános Sender paraméterre való hivatkozás volt, a megfelelő adattípusra (ami jelen esetben TEdit) való átalakítással. Míg egy ilyen eljárást egy megegyező típusú komponenshez kötünk, nem lehet probléma. Máskülönben azonban számos típusellenőrzést kellene végrehajtani (az „is" operátor használatával), mely valószínűleg olvashatatlanabbá tenné a kódot. Véleményem szerint az eljárások megosztása csak azonos típusú vezérlők között ajánlott.

Ahhoz, hogy a felhasználónak megmondjuk, melyik szerkesztő doboz tartalmaz hibás szöveget, az egyes Edit komponensek Tag tulajdonságának értéket kellett adnunk 1-től 5-ig.

Kifinomult beviteli sémák

Az utolsó példában láthattuk, hogyan lehet speciális beviteli célokhoz alakítani az Edit komponenst. Régebben a komponensek valójában csak számokat fogadhattak el, de összetett beviteli sémák kezelése ilyen eszközökkel nem volt szerencsés. Ilyen okból a Borland egy kész maszkolt szerkesztőt készített, melynél a beviteli maszkot egy karakterláncban tárolták.

Például az öt jegynél nem nagyobb számok kezeléséhez az EditMask tulajdonságot 99999-re kell állítani (a 9-es mint nem kötelező számjegy szerepel; lásd a Delphi dokumentációját a szerkesztő maszkban használható különböző karakterekről és szimbólumokról). A string közvetlen begépelése helyett ajánlott inkább a mezőben lévő kis gombon való kattintással megnyitni a hozzá tartozó szerkesztőt. Az Input Mask szerkesztő tesztablakot, és néhány gyakran használt beviteli értéket is tartalmaz (lásd 8.11. ábra).



8.11. ábra *A MaskEdit komponens EditMask tulajdonságának módosítása*

Vegyük észre, hogy az Input Mask szerkesztőben mi is megadhatunk egy maszkot, de ugyanakkor meg kell határoznunk az input részére fenntartott karakterhelyeket, és hogy a maszkban szereplő literálokat elmentse-e a rendszer a végső karakterláncba. Megadhatjuk például, hogy egy telefonszámnál a körzetszámot körülvevő zárójelek csak a bevitelt segítsék, vagy megjelenjenek a tárolt számban is. Az Input Mask szerkesztő ezen két értéke a maszk két utolsó mezőjének (alapértelmezés szerint pontosvesszővel kell őket elválasztani) felel meg.

További beviteli maszkok megtekintéséhez a Masks gombra való kattintással maszk állományokat nyithatunk meg. Az adott fájlok ország szerint csoportosított standard kódokat tartalmaznak. Egy olasz csoport megnyitásakor például megtalálhatjuk az adófizető számot (vagy a pénzügyi számot, ami az amerikai

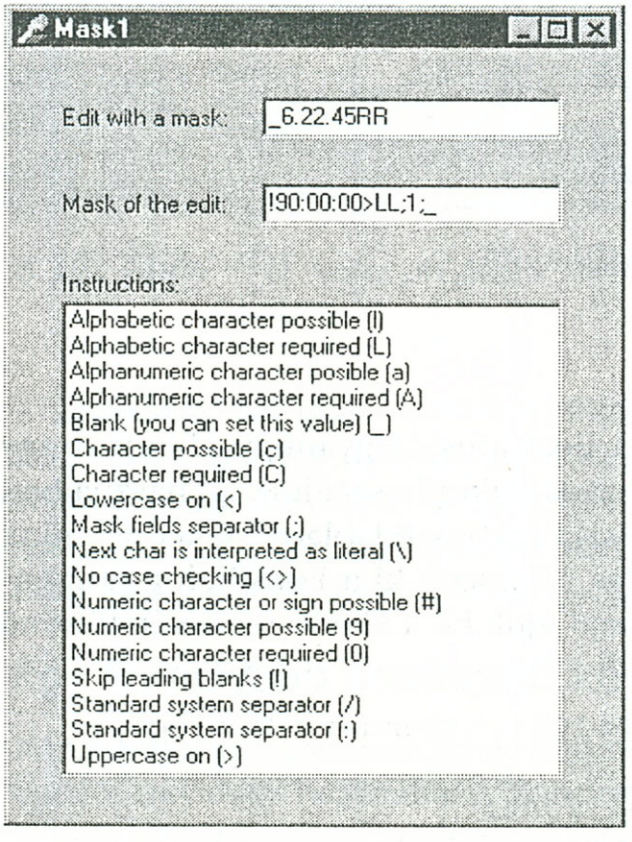
társadalombiztosítási szám megfelelője). Ez a kód betűk és számok bonyolult keveréke (elkészítéséhez a nevet, a születési dátumot és egyéb adatot is felhasználnak) , ahogy a maszk is mutatja:

```
LLLLLL00L00L000L
```

Az ilyen kódban L szerepel a betűk, míg 0 a számok helyén. A Súgóban mindez megtalálható, az alábbi Mask1 példa (8.12. ábra) pedig egy összegzést ad ezekről a kódokról.

A példa formja tartalmaz egy MaskEdit és egy Edit komponenst. Az Edit a másik komponens EditMask tulajdonságának futásidőben történő módosítására használható. Ennek megvalósításához csak néhány sort kellett írni, melyek indításkor (az OnCreate esemény) bemásolják a tulajdonság szövegét a szerkesztőbe, illetve ugyanezt fordítva, valahányszor a mező módosul (Edit1Change):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Edit1.Text := MaskEdit1.EditMask;
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
  MaskEdit1.EditMask := Edit1.Text;
end;
```



8.12. ábra *A Mask1 program kimenetének egy példája*

Egy egyszerű szerkesztő készítése

Az Edit komponensek csak egy korlátozott nagyságú szöveget tudnak kezelni, és csak egyetlen sort. Hosszabb szöveges bemenet fogadására a Memo (feljegyzés) komponenst kell használni. A feljegyzés olyan, mint a szerkesztőmező, viszont több sor befogadására képes, és gördítősávokkal rendelkezik a szövegben való mozgásra.

A Memo-t legkönnyebben szövegszerkesztőként tudjuk használni, ezt illusztrálja Notes nevű következő példánk. Lényege egy szerkesztő létrehozása, amely teljes egészében lefedi saját ablakát (vagy formját), a Windows saját Jegyzettömbjéhez hasonlóan. Az egyetlen általunk nyújtott további lehetőség a szerkesztő karakterkészletének módosítása.

Mindkét rész megvalósítása meglehetően egyszerű. Készítsünk egy új projektet és helyezzünk a formra egy Memo komponenst. Töröljük ki a szövegét, távolítsuk el a keretet, és állítsuk az Alignment tulajdonságát alClient-re, így az lefedi a form teljes belső felületét. Ezen kívül a Memo ScrollBars tulajdonságának ssBoth értékre állításával tegyük rá a két gördítősávot, a vízszinteset és a függőlegeset. Íme az eredmény:

```
object NotesForm: TNotesForm
  Caption = 'Notes'
  object Memo1: TMemo
    Align = alClient
    BorderStyle = bsNone
    Font.Height = -19
    Font.Name = 'Times New Roman'
    ScrollBars = ssBoth
    OnDblClick = Memo1DblClick
  end
  object FontDialog1: TFontDialog...
end
```

A Font párbeszédablak

A program második része tartalmazza a betűtípust. Ugyanúgy ahogyan egy előző példánkban a standard szín párbeszédablakot használtuk, itt használhatjuk a Windows beépített betűtípus kiválasztó párbeszédablakát. Csak menjünk a Components paletta Dialogs oldalára és válasszuk ki a FontDialog komponenst. Helyezzük el valahol a formon, majd írjuk be a Memo területén történő dupla kattintáshoz tartozó kódot:

```
procedure TNotesForm.Memo1DblClick(Sender: TObject);
begin
  FontDialog1.Font := Memo1.Font;
  if FontDialog1.Execute then
    Memo1.Font := FontDialog1.Font;
end;
```

Ez a programrész az aktuális betűkészletet eltárolja a megfelelő tulajdonságban, így alapértelmezés szerint azt választja ki, majd megjeleníti az adott párbeszédablakot (lásd 8.13. ábra). Ennek végeztével a Font tulajdonság a felhasználó által kiválasztott betűtípust tartalmazza, melyet az OK gomb megnyomása esetén a fenti kód harmadik sora bemásol a Memo komponensbe.

8.13. ábra *A Notes program Font párbeszédablaka*

A következő program sokkal hatékonyabb, mint azt első ránézésre hinnénk. Lehetővé tesz például másolási és beillesztési műveleteket a billentyűzet segítségével – ami azt jelenti, hogy kedvenc szövegszerkesztőnkből egy részt belemásolhatunk – és a betűkészlet színét is tudjuk kezelni. Miért ne használhatnánk egy nagy és színes üzenet megjelenítésére a képernyőn?

Egy komolyabb szerkesztő készítése

Bár a Notes programban lehetőség nyílt a betűkészlet kiválasztására, a szöveg egésze ugyanazt használta. A Windows 95 egy új vezérlővel rendelkezik, mely kezeli a Rich szövegformátumot (Rich Text Format, RTF). A Delphi új RichEdit komponense magába foglalja ennek a standard vezérlőnek a viselkedését.

A Delphihez csatolt példák között található egy teljes szerkesztő, amely a RichEdit komponenst használja (a példa neve szintén RichEdit). Mi azonban csak némiképp módosítjuk az előző programunkat, a Memo komponens RichEditre cserélésével, és a kiválasztott rész, nem pedig a teljes szöveg betűtípusának megváltoztathatóságával.

A RichNote példa egy a teljes munkaterületet kitöltő RichEdit komponenst tartalmaz, amely azonban nem rendelkezik duplakattintás eseménnyel. Ehelyett egy betűtípus váltására alkalmas nyomógomb szerepel a form tetejéhez igazított panelben, ami így egy nagyon egyszerű eszköztárat eredményez. Íme a három komponens néhány tulajdonságának szöveges leírása:

```
object RichEdit1: TRichEdit
  Align = alClient
  HideScrollBars = False
  ScrollBars = ssBoth
end
object Panel1: TPanel
  Align = alTop
  object Button1: TButton
    Caption = '&Font...'
  end
end
```

Figyeljük meg a gomb címkéjét, mely egy & jelet tartamaz a billentyűparancs meghatározására, és három pontot a végén, ami azt jelzi, hogy egy párbeszédablakot nyit meg. Ha a felhasználó egy szövegrész kijelölése után a gombra kattint, a program a standard Font ablakot mutatja, kezdeti értéknek a RichEdit komponens betűtípusát véve. Ennek végrehajtása után a kiválasztott betűtípus bekerül az aktuális elem tulajdonságába. A RichEdit komponens DefAttributes és SelAttributes tulajdonságai ugyan nem TFont típusúak, de azzal kompatíbilisek, így az Assign eljárás használatával bemásolhatjuk az értéket:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
  begin
    FontDialog1.Font.Assign(RichEdit1.DefAttributes);
    if FontDialog1.Execute then
      RichEdit1.SelAttributes.Assign(FontDialog1.Font);
  end
  else
    ShowMessage ('No text selected');
end;
```

A RichEdit más betűtípusokhoz és bekezdésformázáshoz kapcsolódó tulajdonságokat is tartalmaz, a fenti kód azonban elég ahhoz, hogy a felhasználók a Memo lehetőségeihez képest sokkal összetettebb szöveget készíthessenek. A könyv további programjaiban egyébként még előfordul a RichEdit komponens.

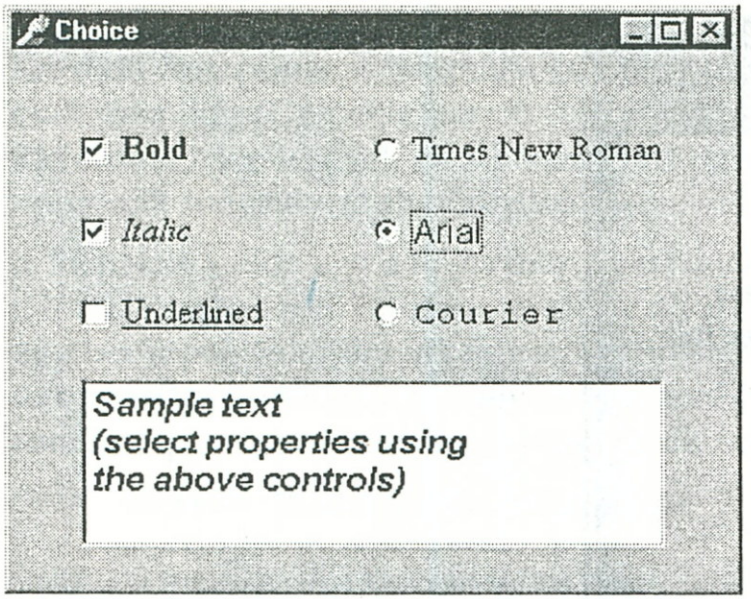
8.14. ábra *A RICHNOTE programmal elérhető font hatások példája*

Választások

A felhasználó két standard Windows vezérlő segítségével különböző lehetőségek közül választhat. Az első a jelölőnégyzet, mely egy szabadon kijelölhető opcióhoz kapcsolódik (amíg ki nem kapcsolják), a második pedig a rádiógomb, amellyel egymást kizáró lehetőségek közül választhatunk. Ha például két rádiógomb az A és B címkékkel rendelkezik, az A vagy a B kiválasztható, de egyszerre mindkettő nem. Másik sajátsága ennek a típusú vezérlőnek, hogy egyet azonban mindenképpen ki kell választani a rádiógombok közül.

Ha a jelölőnégyzetek és rádiógombok közötti különbség még mindig nem teljesen érthető, egy példa segíthet. A 8.15. ábrán a Choice példa eredménye látható. Három jelölőnégyzet található a Bold (félkövér), Italic (dőlt) és az Underlined (aláhúzott) stílusok kiválasztására, illetve három rádiógomb a betűtípus (Times New Roman, Arial vagy Courier) meghatározására. Ezen kívül szerepel még egy Memo komponens néhány sor szöveggel, mely a felhasználó által kiválasztott értékek hatásának azonnali bemutatására szolgál.

A jelölőnégyzetek és rádiógombok használati célja közötti különbség nyilvánvaló. A szöveg lehet félkövér, dőlt és aláhúzott egyszerre, viszont Arial és Courier egy időben nem. A felhasználónak egyetlen betűtípust kell kiválasztania (de egyet biztosan), viszont az egyes stílusokat a másik kettőtől függetlenül állíthatja (esetleg egyiket sem választja).



8.15. ábra *A Choice példa eredménye*

A program néhány egyszerű programsorból áll. A jelölőnégyzeten vagy a rádiógombokon való kattintáskor a megfelelő hatást kell elérnünk. A szövegstílusoknál meg kell néznünk a vezérlő Check (Kijelölt) tulajdonságát, és az adott elemet hozzá kell adnunk a Memo Font (Betűtípus) tulajdonságának Style (Stílus) beállításához, vagy el kell távolítanunk onnan:

```
procedure TForm1.CheckBoldClick(Sender: TObject);
begin
  if CheckBold.Checked then
    Memo1.Font.Style := Memo1.Font.Style + [fsBold]
  else
    Memo1.Font.Style := Memo1.Font.Style - [fsBold];
end;
```

A másik két jelölőnégyzet OnClick eseményének kódja hasonló. A rádiógombok alapkódja még egyszerűbb, mivel rákattintással nem lehet azokat kikapcsolni:

```
procedure TForm1.RadioTimesClick(Sender: TObject);
begin
  Memo1.Font.Name := 'Times New Roman';
end;
```

A rádiógombok csoportosítása

A rádiógombok egymást kizáró választási lehetőségeket képviselnek, egy form azonban tartalmazhat különböző rádiógomb csoportokat. A Windows nem ismerheti a különböző gombok logikai kapcsolatát, ezért szükség van egyfajta tartalmazó komponensre, mely az egy csoportba tartozó rádiógombokat gyűjti össze. A Windows-ban és a Delphi-ben egyaránt ez a megoldás található. A

standard Windows felhasználói interfész egy csoportosítót (group box) használ a rádiógombok összetartásához, funkcionálisan és vizuálisan egyaránt. A Delphiben ezt a vezérlőt a GroupBox komponenssel implementálták. Emellett a Delphinek van egy másik komponense, melyet speciálisan a rádiógomboknál lehet használni: a RadioGroup (rádiócsoport) komponens. Ez egy csoportosító, néhány megjelenített rádiógomb utánzattal. Az utánzat alatt azt értem, hogy ténylegesen a RadioGroup komponens egy önálló vezérlő, mely saját felületére rajzol rádiógombokhoz hasonló elemeket, és nem hivatkozik más hozzá rendelt vezérlőre.

A rádiócsoport használata egyszerűbb ugyan a csoportosítónál, én azonban a hagyományosabb megközelítést használom annak demonstrálására, hogy miképpen lehet olyan kódot írni, mely egy másik vezérlőben elhelyezett vezérlőkkel dolgozik. Maga a tény, hogy néhány vezérlő egy másikba van ágyazva, még egy indok ezen módszer elvetésére, mert elegünk lehet a képernyőn lévő több ablakból, melyek pazarolják a rendszer erőforrásait és valamivel lassabb kódot eredményeznek. Ezen kívül a RadioGroup komponens automatikusan rendezi rádiógombjait, így azt menet közben könnyen bővíthetjük új választási lehetőségekkel. A két megközelítés közötti különbségeket a következő példa mutatja be.

Egy rádiógombokat tartalmazó csoportosító elkészítésének szabályai nagyon egyszerűek. Helyezzük a GroupBox komponenst a formra, majd tegyük bele a rádiógombokat. A GroupBox más komponenseket is tartalmazhat, és a Panel mellett egyike a leggyakrabban használt tartalmazó komponenseknek. A csoportosító kikapcsolásával vagy elrejtésével a benne lévő vezérlők is elérhetetlenek lesznek.

Kezelhetjük ezután külön-külön a rádiógombokat, de hozzáférhetünk a csoportdoboz ezen vezérlőket tartalmazó tömbjéhez is. Mint azt a legutóbbi fejezetből már tudjuk, az erre a tömbre utaló tulajdonság a Controls. A tömb méretét egy másik tulajdonság, a ControlCount tartalmazza, melyet az előzőhöz hasonlóan csak futás közben érhetünk el.

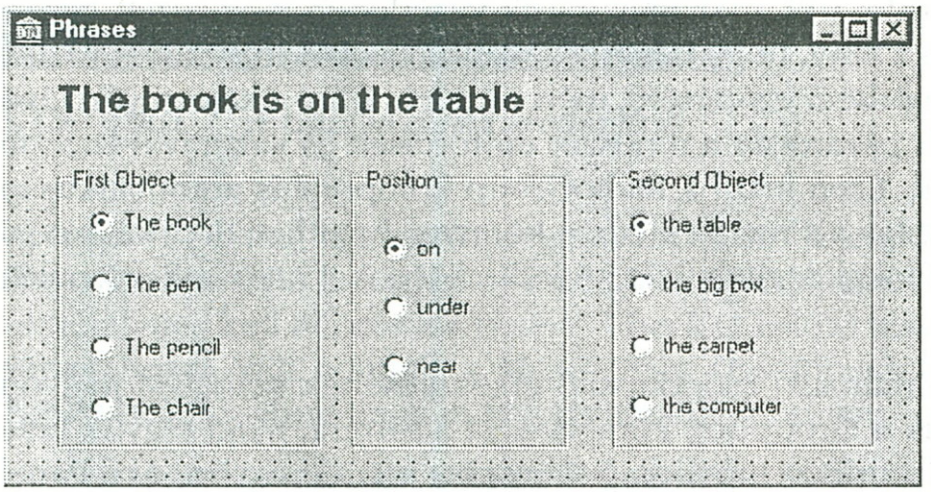
A Phrases1 példa

Egy idegen nyelv elsajátításakor az ember bizonyára eltölt némi időt suta és haszontalan kifejezések ismételgetésével. Angol tanulása közben az egyik legjellemzőbb a hírhedt „The book is on the table." példamondat. A rádiógombok bemutatása céljából a Phrases1 példában különböző lehetőségek kiválasztásával hasonló kifejezéseket állíthatunk össze. A formot a 8.16. ábrán láthatjuk.

A form meglehetősen összetett. Elkészítése során ügyeljünk arra, hogy először a GroupBox komponenst kell elhelyezni, és a rádiógombokat csak ezt követően. Ezek után minden elem címkéjét megfelelően be kell írni, ahogyan az a 8.16. ábrán látható. Az utolsó kijelölés igazából egy RadioGroup komponens,

nem pedig néhány rádiógombot tartalmazó csoportosító (mint az a form alább található szöveges leírásából is kiderül). Ilyen esetekben az egyes választási lehetőségeket az Items tulajdonságban kell felsorolni.

Ne feledjünk létrehozni címkéket, kiválasztani az azoknak megfelelő nagyobb betűkészletet, és begépelni a megfelelő szöveget a tervezés közben bekapcsolt gomboknál. Ez egy fontos dolog: ha egy formban, vagy egy csoportban elhelyezünk néhány rádiógombot, ne feledkezzünk el arról, hogy egyet bekapcsoljunk tervezés közben. Csoportonként egy gombot mindenképpen be kell jelölni, és a rádiócsoport ItemIndex tulajdonságának, mely az aktuális választást jelzi, ennek megfelelő értéket kell tartalmaznia.



8.16. ábra *A Phrases1 példa formja*

Összegzésképpen íme a form kódja:

```
object Form1: TForm1
  Caption = 'Phrases'
  object Label1: TLabel
    Width = 243
    Caption = 'The book is on the table'
    Font.Height = -21
    Font.Name = 'Arial'
    Font.Style = [fsBold]
  end
  object GroupBox1: TGroupBox
    Caption = 'First Object'
    object RadioBook: TRadioButton
      Caption = 'The book'
      Checked = True
      OnClick = ChangeText
    end
    object RadioPen: TRadioButton
```

```
        Caption = 'The pen'
        OnClick = ChangeText
      end
      object RadioPencil: TRadioButton...
      object RadioChair: TRadioButton...
    end
    object GroupBox2: TGroupBox
      Caption = 'Position'
      object RadioOn: TRadioButton
        Caption = 'on'
        Checked = True
        OnClick = ChangeText
      end
      object RadioUnder: TRadioButton...
      object RadioNear: TRadioButton...
    end
    object RadioGroup1: TRadioGroup
      Caption = 'Second Object'
      Items.Strings = (
        'the table'
        'the big box'
        'the carpet'
        'the computer')
      OnClick = ChangeText
    end
  end
```

Most egy kis programot kell írnunk, hogy a rádiógombon való kattintás a kifejezés megfelelő megváltozását eredményezze. Ezt több módon érhetjük el, például az utolsó példához hasonlóan minden gomb OnClick eseményéhez egy-egy eljárást rendelünk, és a kifejezés különböző részeit a form más-más változójában tároljuk. Módosítjuk a gombnak megfelelő szövegrészt, majd újraépítjük a mondatot.

Egy másik megoldás szerint egyetlen eljárást írunk, ami megnézi, hogy éppen melyik gombot jelöltük be, és a beállításokkal összhangban felépíti a kifejezést. Ezt az eljárást az egyes gombok és a RadioGroup komponens OnClick eseményéhez kell kapcsolni, ami nem nehéz feladat. Válasszuk ki a form összes rádiógombját (a SHIFT billentyű lenyomása mellett kattintsunk mindegyiken), majd írjuk be az eljárás nevét az Object Inspectorba. Mivel az új mondatot összeállító eljárás nem hivatkozik semmilyen különleges vezérlőre, az OnClick esemény után a második oszlopba tetszőleges nevet írhatunk. Íme az önálló összetett eljárás:

```
procedure TForm1.ChangeText(Sender: TObject);
var
  Phrase: string;
  I: integer;
```

```
begin
  {look at which radio button is selected
  and add its text to the phrase}
  for I := 0 to GroupBox1.ControlCount - 1 do
    if (GroupBox1.Controls[I] as TRadioButton).Checked
then
      Phrase := (GroupBox1.Controls[I] as
                  TRadioButton).Caption;

  {add the verb and blank spaces}
  Phrase := Phrase + ' is ';

  {repeat the operation on the second group box}
  for I := 0 to GroupBox2.ControlCount - 1 do
    with GroupBox2.Controls[I] as TRadioButton do
      if Checked then
        Phrase := Phrase + Caption;

  {retrieve the radio group selection, and display
  the result in the label}
  Label1.Caption := Phrase + ' ' +
    RadioGroup1.Items [RadioGroup1.ItemIndex];
end;
```

A ChangeText eljárás először megnézi, hogy a rádiógombok közül az első csoportból melyiket választottunk ki, majd elhelyezi az igét („is") a szavak közti szóközökkel együtt. A csoportban bejelölt vezérlő meghatározására az eljárás egy for ciklust használ a lehetőségek végigpásztázására. Mivel a Controls tömb indexelése nullánál kezdődik a ciklus 0-tól indul a vezérlők száma mínusz 1-ig, és ellenőrzi, hogy a gomb Checked tulajdonsága True (igaz)-e. A művelet végrehajtásához szükséges egy típusátalakítás – egy általános vezérlőn nem használhatjuk a Checked tulajdonságot. A bejelölt rádiógomb megtalálásakor a program egyszerűen a karakterláncba másolja annak címkéjét. Ekkor a for ciklus már akár véget is érhetne, de mivel egyszerre csak egy rádiógomb lehet bejelölve, biztonságosabb megvárni, amíg végignézi az összes elemet, és önmagától leáll. Ugyanaz a művelet ismétlődik kétszer, de a második alkalommal egy „with" utasítás rövidebbé és olvashatóbbá teszi a kódot.

Mint a fenti eljárás végén láthatjuk, a RadioGroup komponens használatával a kód sokkal egyszerűbb. Ez a vezérlő rendelkezik egy ItemIndex tulajdonsággal is, amely az éppen kiválasztott gombot jelöli, és egy Items-el, mi a rádiógombok szövegének listája. Összességében egy rádiócsoport és egy listapanel (list box) használata (ahogyan a következő példánkban látni fogjuk) felhasználói interfészük alapvető különbségeitől eltekintve nagyon hasonlít egymásra.

Egy lista sok választási lehetőséggel

A rádiógombok nem alkalmasak túl sok választási lehetőség felkínálására, vagy legalábbis igen nagy formot kellene használnunk. Általában nem több mint öt, vagy hat értékre szokták használni. Másik probléma, hogy bár egy rádiógombot ki lehet kapcsolni, a csoport elemei általában rögzítettek, csak rádiócsoportok használatakor rendelkezünk némi rugalmassággal. Az iménti problémák megoldása vezet el a listapanel (list box) használatához. A listapanel kis helyen kínál fel sok választási lehetőséget, mivel egyszerre csak a lista egy részét mutatja, és a gördítősáv segítségével juthatunk el a többi lehetőséghez. Másik előnye, hogy a benne lévő elemeket könnyen bővíthetjük, vagy törölhetünk belőlük.

Tipp

A ListBox összetevő használatának még egy fontos jellemzője, hogy választhatunk a rádiógombokhoz hasonló egyszeres választók és a jelölőnégyzetekhez hasonló többszörös-kijelölő mód közül. A példa következő változata egy többszörös-kiválasztó listapanelt mutat.

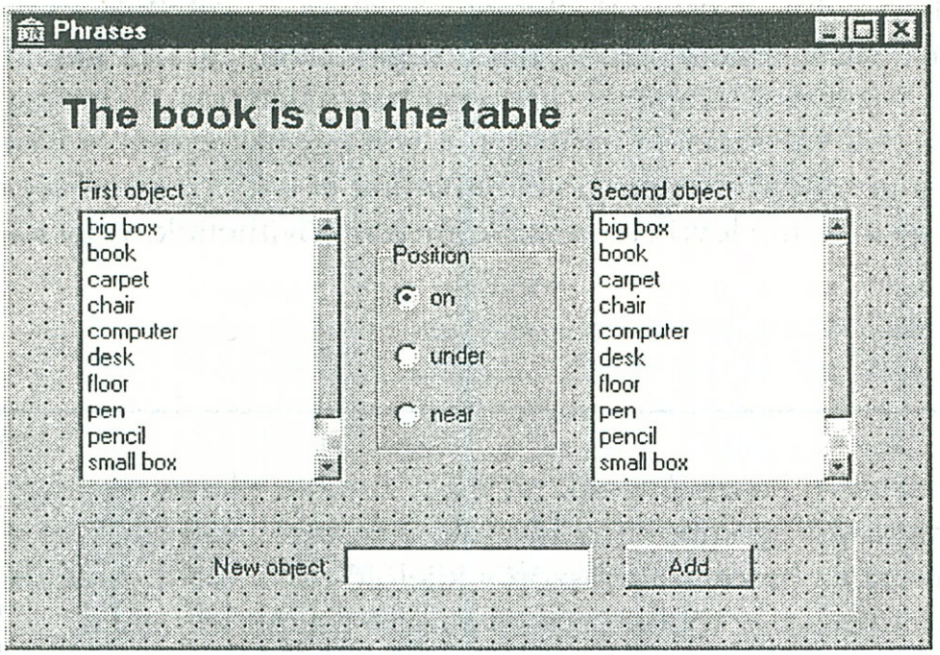
Nézzünk egyelőre egy egyválasztós listát (single-selection list box). A Phrases1 példát némiképp átalakíthatjuk néhány ilyen komponens segítségével. A kifejezés első és második tárgyának kiválasztásához a sok rádiógomb helyett két listát használunk. Amellett, hogy jóval több elemet szerepeltethetünk, még újabbakat is elhelyezhetünk bennük. Elkerülhetjük ezen kívül, hogy ugyanazt a tárgyat kétszer is kiválassza a felhasználó, hiszen ez olyan mondatot eredményez, mint például a „The book is on the book.". Mint az várható, ez a példa az előzőnél sokkal összetettebb, és néhol aránylag bonyolult kódot igényel.

A Phrases2 példa formja

Az első lépés szokás szerint a form felépítése (8.17. ábra). Utolsó példánkból is kiindulhatunk, a két szélső csoport listapanelekkel való helyettesítésével (a csoporton belüli rádiógombok törlése automatikusan megtörténik). A középső csoport helyére egy rádiócsoport került, így tulajdonképpen nem is maradt sok az előző példából.

Ezek után írjunk néhány stringet a listák Items tulajdonságába. A példa helyes működése érdekében a két listapanelnek ugyanazokat a karakterláncokat kell tartalmaznia; egyikük Items tulajdonságának szerkesztőjéből ki tudjuk másolni, majd a másik komponens megfelelő tulajdonságának szerkesztőjébe be

tudjuk illeszteni ezeket. A program hatékonyságának növelése érdekében a Sorted (rendezett) tulajdonságot True-ra állítva elérhetjük a listában szereplő elemek sorba rendezését. Végül ne felejtsünk el tartalmukat jellemző címkéket helyezni az egyes listák fölé.



8.17. ábra *A Phrases példa második változatának formja*

A form alsó részén (8.17. ábra) található még egy címkével ellátott szerkesztőmező, egy gomb, és egy ezeket vizuálisan összefogó keret (ez utóbbi csak egy grafikus elem, nem tartalmazza a rápakolt komponenseket). Mint később kiderül, a felhasználó a gomb lenyomásával mindkét list boxot kibővítheti a mezőben szereplő szöveggel. Ez utóbbi művelet azonban csak akkor hajtandó végre, ha az adott string nem üres, és még nem szerepel a listákban.

Íme a felújított form komponensinek szöveges leírása (mely sokban különbözik az előző verziótól):

```
object Form1: TForm1
  Caption = 'Phrases'
  OnCreate = FormCreate
  object Label1: TLabel... // as in Phrases1
  object Label2: TLabel
    Caption = 'First object'
  end
  object Label3: TLabel
    Caption = 'Second object'
  end
  object ListBox1: TListBox
    Items.Strings = (
```

```
        'big box'
        'book'
        'carpet'
        'chair'
        'computer'...)
      Sorted = True
      OnClick = ChangeText
    end
    object RadioGroup1: TRadioGroup
      Caption = 'Position'
      Items.Strings = (
        'on'
        'under'
        'near')
    end
    object ListBox2: TListBox... // identical to ListBox1
    object Bevel1: TBevel...
    object Label4: TLabel
      Caption = 'New object'
    end
    object EditNew: TEdit...
    object ButtonAdd: TButton
      Caption = 'Add'
      OnClick = ButtonAddClick
    end
  end
```

Listapanelek kezelése

Az form megalkotása után elkezdhetjük írni a kódot. Első teendőnk a két listapanel és a rádiócsoport OnClick eseményéhez kötött új ChangeText eljárás elkészítése, mely az előző példához képest sokkal egyszerűbb. A listában kiválasztott elem megszerzéséhez csupán az adott elem sorszámára van szükségünk (melyet a futási idő alatti ItemIndex tulajdonság tartalmaz), majd ahogyan a Phrases1 program is tette, az Item tömb megfelelő pozíciójáról kiolvashatjuk a kívánt karakterláncot.

Az eljárás kódja kezdetben így néz ki (ez csupán egy átmeneti verzió, mely különbözik a mellékelt CD-n lévőtől):

```
procedure TForm1.ChangeText(Sender: TObject);
var
  Phrase: String;
begin
  Phrase := 'The ';
  Phrase := Phrase + ListBox1.Items
        [ListBox1.ItemIndex];
  Phrase := Phrase + ' is ';
```

```
    Phrase := Phrase + RadioGroup1.Items
          [RadioGroup1.ItemIndex];
    Phrase := Phrase + ' the ';
    Phrase := Phrase + ListBox2.Items
          [ListBox2.ItemIndex];
    Label1.Caption := Phrase;
end;
```

Ez a program azonban nem fog megfelelően működni, mivel az elején egyik listapanelből sem választottunk ki kezdeti értéket. A problémát elkerülhetjük a form OnCreate eseményéhez tartozó kód megírásával, melyben kikeressük a két alapértelmezett értéket, a book-ot és a table-t, majd kiválasztjuk azokat. Ezt két lépesben tehetjük, először az IndexOf eljárás segítségével kiderítjük a string tömb ide tartozó indexét, majd ezt a számot, mint a kiválasztott elem sorszámát szerepeltetjük:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  N : Integer;
begin
  N := ListBox1.Items.IndexOf ('book');
  ListBox1.ItemIndex := N;
  N := ListBox2.Items.IndexOf ('table');
  ListBox2.ItemIndex := N;
end;
```

A kiválasztott string eltávolítása a másik listapanelből

Amint megbirkózunk az előző résszel, újabb két problémával kerülünk szembe. A kiválasztott karakterláncot el kell távolítanunk a másik listából (hogy elkerüljük ugyanannak a szónak kétszeri előfordulását a kifejezésben), és meg kell írnunk a gomb lenyomásához tartozó kódot.

Az első probléma összetettebb, de azonnal részletezni fogom, míg a másodiknak a megoldása részben az először írt kódon alapul. Célunk tehát a komponensből eltávolítani a másikban kijelölt elemet. Ezt nem is lenne nehéz megírni, a probléma azonban az, hogy a kiválasztás megváltozása esetén a törölt elemet vissza kell tennünk a listába, különben az nagyon hamar kiürülne. Megoldhatjuk a gondot, ha a két kiválasztott stringet mindig eltároljuk a form lokális mezőiben, a String1-ben és a String2-ben:

```
type
  TForm1 = class(TForm)
  ...
  private
    String1, String2: String;
  end;
```

Ezután módosítani kell az indításkor, és az új elem kiválasztásakor végrehajtott kódot. A FormCreate eljárásban a két karakterlánc kezdeti értékét kell beállítanunk, majd kivennünk őket a másik listapanelből; az elsőt el kell távolítani a másodikból és fordítva. Mivel a TString osztály Delete eljárásának szüksége van az indexre, az IndexOf függvényt újra használnunk kell a sorszám meghatározására:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  N : Integer;
begin
  String1 := 'book';
  String2 := 'table';

  {delete the selected string from the other list box
  to avoid a double selection}
  ListBox2.Items.Delete (ListBox2.Items.IndexOf
(String1));
  ListBox1.Items.Delete (ListBox1.Items.IndexOf
(String2));

  {select the two strings in their respective list
boxes}
  N := ListBox1.Items.IndexOf (String1);
  ListBox1.ItemIndex := N;
  N := ListBox2.Items.IndexOf (String2);
  ListBox2.ItemIndex := N;
end;
```

Figyelem!

A stringet kiválasztó kódnak a Delete után kell következnie, mert egy elem eltávolítása befolyásolhatja a kiválasztást. A kiválasztás ténylegesen csak egy szám, ami egy karakterláncra utal, nem fordítva, ahogyan az kényelmesebb lenne. Ez egyébként nem a Delphi implementáción, hanem a Windows listapanelek működésén múlik.

A dolgok egy új elem kiválasztásakor válnak igazán bonyolulttá. A ChangeText eljárás eleje pár sorral bővült, melyek azonban csak akkor kerülnek végrehajtásra, ha a kattintás valamelyik listán történt (ne feledjük el, hogy a kódot a csoporthoz is hozzácsatoltuk). Mindkét listánál ellenőrizni kell, hogy változott-e a kiválasztott elem, és ha igen, az előzőleg kiválasztott karakterláncot hozzá kell adni a másik listapanelből, az újat pedig ki kell törölni. Íme a ChangeText eljárás új változata:

```
procedure TForm1.ChangeText(Sender: TObject);
var
  TmpStr: String;
begin
  // if a list box has changed
  if Sender is TListBox then
  begin
    // get the text of the first string
    TmpStr := ListBox1.Items [ListBox1.ItemIndex];
    // if the first one has changed
    if TmpStr <> String1 then
    begin
      // update the strings in ListBox2
      {1.} ListBox2.Items.Add (String1);
      {2.} ListBox2.Items.Delete (
             ListBox2.Items.IndexOf (TmpStr));
      {3.} ListBox2.ItemIndex :=
             ListBox2.Items.IndexOf (String2);
      {4.} String1 := TmpStr;
    end;
    // get the text of the second string
    TmpStr := ListBox2.Items [ListBox2.ItemIndex];
    // if the second one has changed
    if TmpStr <> String2 then
    begin
      // update the strings in ListBox1
      ListBox1.Items.Add (String2);
      ListBox1.Items.Delete (ListBox1.Items.IndexOf
        (TmpStr));
      ListBox1.ItemIndex := ListBox1.Items.IndexOf
        (String1);
      String2 := TmpStr;
    end;
  end;
  // build the phrase with the current strings
  Label1.Caption := 'The ' + String1 + ' is ' +
    RadioGroup1.Items [RadioGroup1.ItemIndex] +
    ' the ' + String2;
end;
```

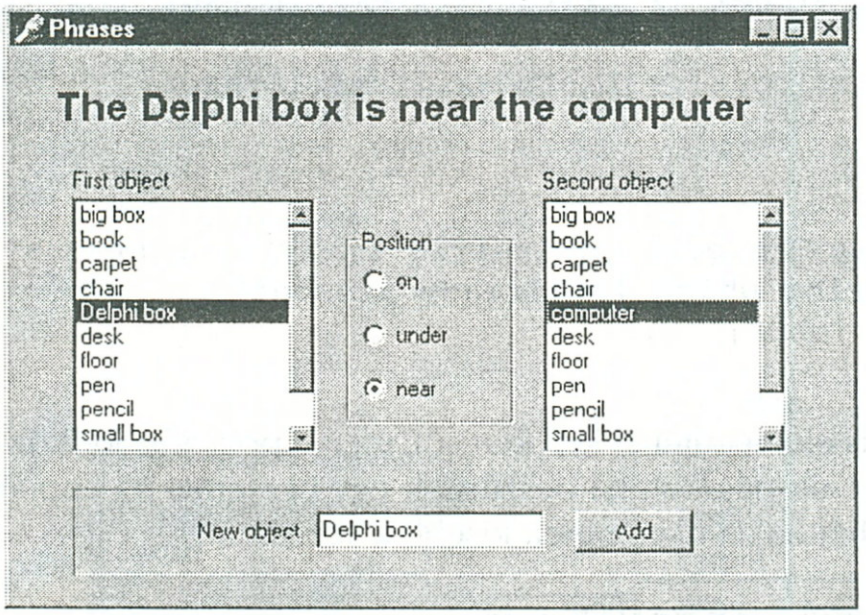
Mit csinál a kód első része? A műveletek alábbi részletezésében az új kiválasztás az első boxban történt. Az első listából kiválasztott elemet az eljárás a TmpStr átmeneti stringben tárolja. Ha ez különbözik a korábbitól, a String1-től, négy művelet kerül végrehajtásra (számozás a fenti programlista szerint):

1. Az előzőleg kiválasztott karakterláncot, a String1-et hozzáadjuk a másik listához, a ListBox2-höz.

2. Az új kiválasztást, a TmpStr-t eltávolítjuk a másik listából.

3. A másik lista kiválasztott elemét, a String2-t újra kiválasztjuk a listából arra az esetre, ha esetleg az előző utasítások hatására módosult volna a pozíciója.
4. Miután a két listapanel megfelelő elemeket tartalmaz, beállíthatjuk a String1 új értékét, melyet majd a mondat összeállításakor fogunk használni.

Pár sorral később ugyanezeket a műveleteket végezzük el a másik listapanelen is. Vegyük észre, hogy a kifejezés összeállításához már nincs szükségünk az egyes listákra, mert a kívánt értékek már a String1 és String2 változókban vannak. Az eredmény a 8.18. ábrán látható.



8.18. ábra *A Phrases2 példa. Vegyük észre, hogy az egyik listapanelben kiválasztott string nem szerepel a másikban*

Az Add nyomógomb OnClick eseményének megírása már nagyon egyszerű. Egyedül annak ellenőrzésére kell ügyelnünk, hogy ténylegesen létezik-e szöveg a szerkesztő mezőben, és hogy a két listában szerepel-e már a beírt karakterlánc. Ha ezt csak az egyiknél ellenőriznénk, akkor a másikban kiválasztott elem és a szerkesztőmezőben lévő szöveg elvesztené kapcsolatát.

Ellenőrzéskor mindkét listadoboztól megkérdezzük az új string sorszámát, ami ha az nem szerepel a listában, a -1 értéknek felel meg. Máskülönben az IndexOf függvény visszatér a helyes sorszámmal, ami az első elemnél 0. Szaknyelven szólva azt mondhatjuk, hogy a függvény az adott elem nulla-bázisú sorszámát adja vissza, vagy a -1 hibakódot, ha nem találja meg. Íme a kód:

```
procedure TForm1.ButtonAddClick(Sender: TObject);
begin
  {if there is a string in the edit control and
  the string is not already present in one of the
lists}
  if (EditNew.Text <> '') and
    (ListBox1.Items.IndexOf(EditNew.Text) < 0) and
    (ListBox2.Items.IndexOf(EditNew.Text) < 0) then
  begin
    {add the string to both list boxes}
    ListBox1.Items.Add (EditNew.Text);
    ListBox2.Items.Add (EditNew.Text);
    {reselects the current items properly}
    ListBox1.ItemIndex := ListBox1.Items.IndexOf
(String1);
    ListBox2.ItemIndex := ListBox2.Items.IndexOf
(String2);
  end
  else
    MessageDlg ('The edit control is empty or contains'
      + ' a string which is already present',
      mtError, [mbOK], 0);
end;
```

Az eljárás végén az aktuális elemet újra ki kell keresni, mert a pozíciója időközben megváltozhat. Ez történik például, ha beszúrunk egy új elemet az éppen kiválasztott string elé – tehát ha az új elemnek kisebb a sorszáma.

Többszörös kiválasztás engedélyezése

Egy listadobozban megadhatjuk, hogy csak egy elemet, vagy akár többet is ki lehessen választani. Döntésünket a Multiple (többszörös) tulajdonság beállításával érvényesíthetjük. Mint a neve is mutatja, a True (igaz) érték esetén enged meg többszörös kijelölést. A Windows és a Delphi list boxaiban valójában ennek is két típusa van: a többszörös kijelölés és a bővebb kijelölés. Az első esetben a felhasználó egyszerűen az elemeken való kattintással választhatja ki a kívántakat, míg a másodiknál a SHIFT és a CTRL billentyűkkel lehet kiválasztani az egymást követő ill. az egymástól távolabbi elemeket. Ez utóbbit határozza meg az ExtendedSelect tulajdonság.

Míg egy többszörös-kiválasztó listapanel beállítása nagyon egyszerű, a kód írásánál már problémák jelentkeznek. Egy egyválasztós lista kiválasztott elemének lekérdezése nagyon könnyű, az ItemIndex tulajdonság tartalmazza annak indexét, ami alapján egy egyszerű kifejezés már megadja a szöveget:

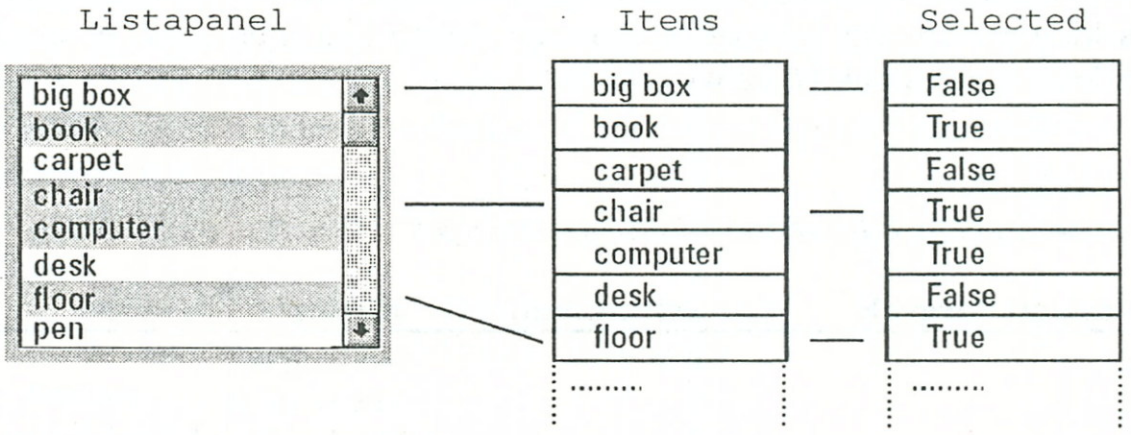
```
ListBox2.Items[ListBox2.ItemIndex];
```

Egy többszörös-kiválasztó listapanel esetén azonban nem tudjuk a kijelölt elemek számát, még azt sem, hogy egyáltalán van-e olyan. A felhasználó valójában a lista bármelyik pontján kattintva az adott elemet, a mutató húzásával pedig egymást követő elemeket választhat ki, míg a CTRL billentyű lenyomása melletti kattintás a többitől függetlenül befolyásolja a kijelölési állapotot. Ezek segítségével megtehetjük, hogy akár a lista egyik elemét sem választjuk ki.

Az éppen kiválasztott elemekről a program a Selected (kiválasztott) tömb megvizsgálásával kaphat információt. Ez egy logikai értékeket tartalmazó tömb, melynek elemszáma megegyezik a listában szereplő elemek számával. Ahogyan a 8.19. ábra vázlata is mutatja, minden érték azt jelzi, hogy a neki megfelelő elem ki van-e választva.

Egy listában a kiválasztott elemek számának meghatározásához például végig kell nézni a Selected tömböt, általában egy nullától az elemek száma mínusz egyig terjedő ciklussal:

```
SelectCount := 0;
for ListItem := 0 to ListBox1.Items.Count - 1 do
  if ListBox1.Selected[ListItem] then
    Inc (SelectCount);
```



8.19. ábra *Egy többszörös kiválasztást lehetővé tevő listapanel Selected és Items tulajdonságai közötti kapcsolat sémája*

A fenti kódban kiszámított érték pontos meghatározására a ListBox komponens igazából rendelkezik egy nem dokumentált SelCount tulajdonsággal. A következő példában a fenti kód egy bonyolultabb változatát használjuk.

A Phrases példa harmadik változata

Az iménti információk birtokában összeállíthatjuk a Phrases példa egy újabb változatát, melyben a felhasználó több elemet is kiválaszthat az első listából. Az egyedüli különbség az új és a legutóbbi változatok formja között az első listapanel MultiSelect tulajdonságának értéke.

Az form felső részén lévő címke WordWrap (szótörés) tulajdonságának bekapcsolásával és az AutoSize (automatikus méret) kikapcsolásával helyet biztosíthatunk hosszabb mondatok számára. Mivel a példa kódja már így is elég összetett, nem szerepel benne a másik listában kiválasztott elem törlése, mely többszörös kijelölések esetén valóban nagyon bonyolult lenne.

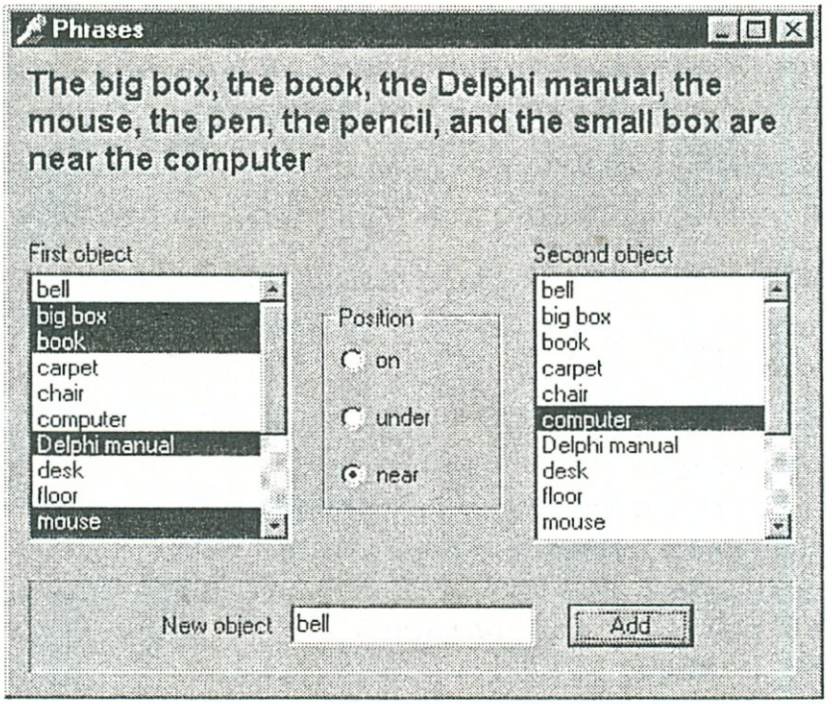
A fő probléma, amellyel szembe kell néznünk, a különböző kifejezéseket mondattá építése. Az alapötlet az, hogy végignézzük a Selected tömböt, és közben a mondathoz csatoljuk a kiválasztott tárgyakat. Az utolsó szó elé azonban el kell helyezni egy „and"-et, és ha csak két elemről van szó, el kell távolítani a vesszőt. Továbbmenve, különbséget kell tennünk egyes- és többesszám között („is" vagy „are"), és meg kell határozni egy alapértelmezett szöveget arra az esetre, amikor egyik szó sincs kijelölve. Amint az a 8.1. táblázatban (és a 8.20. ábrán) látható, az ilyen mondatok felépítése nem olyan egyszerű. Ha például rögzítettük a „The book and the computer," kifejezést, majd szeretnénk kiegészíteni egy harmadik elemmel, vissza kell mennünk, és módosítanunk kell a már meglévő szöveget.

8.1. táblázat

A Phrases3 példa különböző lehetséges stringjei

Kiválasztott Tárgyak	SelectCount	Kifejezés
{none}	0	Nothing is
book	1	The book is
book, computer	2	The book and the computer are
book, computer, pen	3	The book, the computer, and the pen are
book, computer, pen, small box	4	The book, the computer, the pen, and the small box are

Az itt közölt megoldásban két különböző kifejezést készítünk, az egyik arra az esetre szolgál, ha nem adunk hozzá több elemet, a másik pedig későbbi elemek fogadására (az „and" nélkül) jó. A kódban szereplő TmpStr1 string a végleges mondat egy átmeneti változata, míg a TmpStr2-t további elemek fogadására használjuk. A ciklus végére a TmpStr1 a megfelelő értéket tartalmazza. Mint az alábbi kódban is látható, ha csupán két elemről van szó, a kifejezés felépítése némiképp különbözik, mivel az és (and) kapcsolat előtti vesszőt el kell távolítani.



8.20. ábra *A Phrases3 kimenetének egy példája*

Figyeljük meg, hogy egy rendezett lista vizsgálatakor az eredményben az objektumok ábécé sorrendben jelennek meg, nem pedig a kiválasztás sorrendje szerint. A ChangeText eljárás új verziójának alábbi kódján megfigyelhetjük ennek a jelenségnek a megvalósulását (a 8.2. táblázat lépésről lépésre leírja a karakterláncok felépülését):

```
procedure TForm1.ChangeText(Sender: TObject);
var
  Phrase, TmpStr1, TmpStr2: String;
  SelectCount, ListItem: Integer;
begin
  SelectCount := 0;
  {look at each item of the multiple selection list box}
  for ListItem := 0 to ListBox1.Items.Count - 1 do
    if ListBox1.Selected [ListItem] then
    begin
      {if the item is selected increase the count}
      Inc (SelectCount);
      if SelectCount = 1 then
      begin
        {store the string of the first selection}
        TmpStr1 := ListBox1.Items.Strings [ListItem];
        TmpStr2 := TmpStr1;
      end
      else if SelectCount = 2 then
      begin
```

```
            {add the string of the second selection}
            TmpStr1 := TmpStr1 + ' and the ' +
              ListBox1.Items.Strings [ListItem];
            TmpStr2 := TmpStr2 + ', the ' +
              ListBox1.Items.Strings [ListItem];
          end
          else // SelectCount > 2
          begin
            {add the string of the further selection}
            TmpStr1 := TmpStr2 + ', and the ' +
              ListBox1.Items.Strings [ListItem];
            TmpStr2 := TmpStr2 + ', the ' +
              ListBox1.Items.Strings [ListItem];
          end;
        end;
      {build the first part of the phrase}
      if SelectCount > 0 then
        Phrase := 'The ' + TmpStr1
      else
        Phrase := 'Nothing';
      if SelectCount <= 1 then
        Phrase := Phrase + ' is '
      else
        Phrase := Phrase + ' are ';
      {add the text of the radio button}
      Phrase := Phrase +
        RadioGroup1.Items [RadioGroup1.ItemIndex];
      {add the text of the second list box}
      Phrase := Phrase + ' the ' +
        ListBox2.Items [ListBox2.ItemIndex];

      Label1.Caption := Phrase;
    end;
```

8.2. táblázat

A stringek felépítése lépésről-lépésre

Kiválasztott elemek	Lépések	TmpStr1 (a végső kijelentés)	TmpStr2 (az ideiglenes kijelentés)
book	1	book	book
+ computer	2	book and the computer	book, the computer
+ pen	3	book, the computer, and the pen	book, the computer, the pen
+ small box	4	book, the computer, the pen, and the small box	book, the computer, the pen, the small box

A program más eljárásai csak kis mértékben változnak, a FormCreate például egyszerűsödni fog, mivel nem kell törölnünk a kiválasztott elemet a másik listából. Az Add eljárás szintén egyszerűsödik, mert mindkét lista ugyanazokkal az elemekkel rendelkezik, és mert a többszörös-kiválasztó listapanel (multiple-selection list box) nem okoz problémát új elem hozzávétele esetén.

A többszörös-kiválasztó listák állapotának kezelésére az ItemIndex tulajdonság is használható, mely a fókuszt birtokló elem sorszámát tartalmazza. Ha a felhasználó a CTRL lenyomása mellett elemekre kattint, az egyes kattintásoknál megtudhatjuk, hogy éppen melyik elemet kapcsolta ki, vagy be – a Selected tömb ennek megfelelő helyén lévő érték lekérdezésével könnyen meghatározhatjuk, hogy a két művelet közül melyik történt. Probléma azonban, hogy több elem húzással történő kijelölése esetén ez az eljárás nem működik. Ez esetben a húzó eljárásokat is befolyásolni kell, ami sokkal összetettebb a korábban leírt technikánál.

A jelölőnégyzet-lista (CheckListBox) komponens használata

A Phrases példa egy további kiegészítése a Borland Delphi 3-ban bevezetett új komponensének, a jelölőnégyzet-listának (CheckListBox) a használata. Ez tulajdonképpen egy listapanel egy speciális megjelenéssel (szaknyelv szerint egy owner-draw list box). A lista minden elemét egy-egy jelölőnégyzet előzi meg. A felhasználó kiválaszthatja a lista egy adott elemét, de azok állapotát a jelölőnégyzeteken kattintva is befolyásolhatja.

Az AllowGrayed (szürke engedélyezése) tulajdonság igaz értéke esetén az egyes négyzetek lehetnek jelöletlenek, szürkék, és kiválasztottak. A jelölőnégyzeten való kattintáskor ez a három állapot váltakozik. Az egyes állapotok lekérdezésére a Checked (kiválasztott) és a State (állapot) tulajdonságok használatosak, mindkettő tömb típusú. A Checked logikai típusú tulajdonság az AllowGrayed False (hamis) értéke esetén használható. A másik, a State egy TCheckBoxState típusú változó:

```
type
  TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed)
```

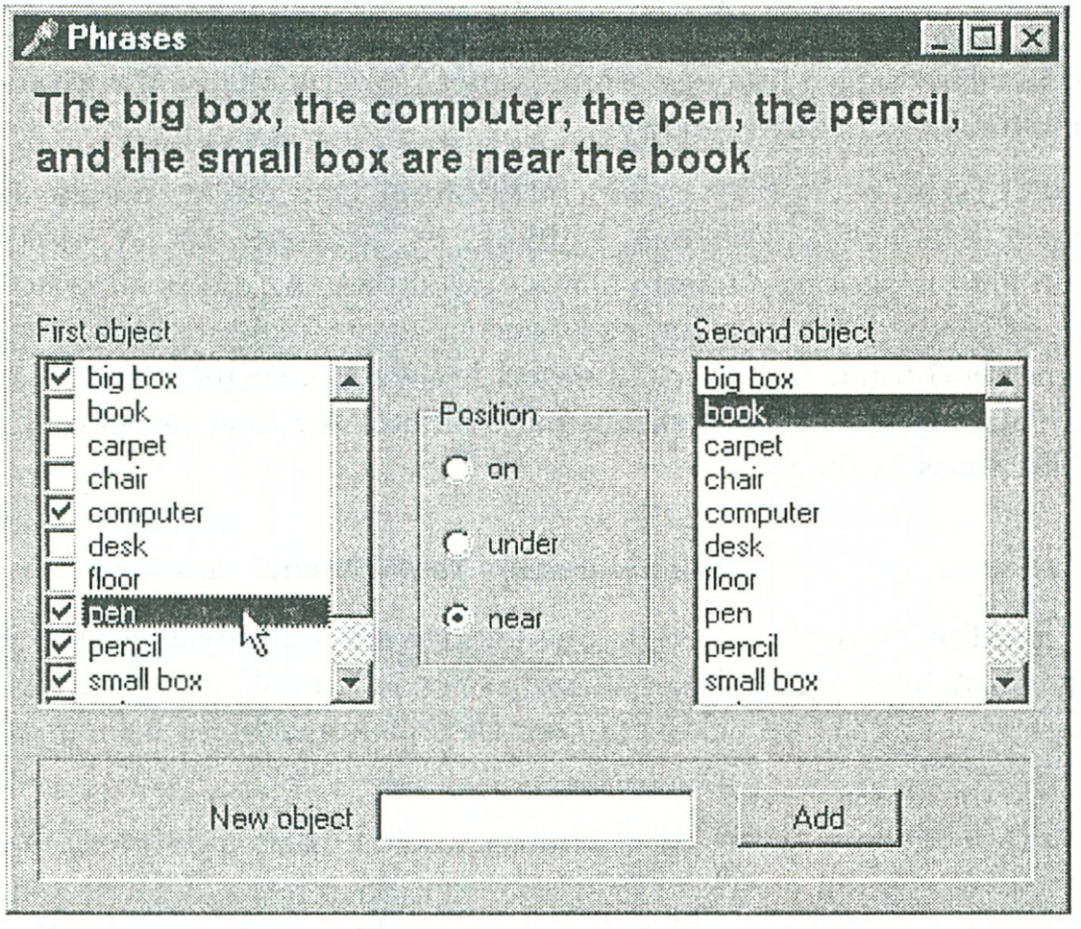
Ez a jellemző az AllowGrayed igaz értéke mellett üzemel, különbséget téve az egyes elemek három lehetséges állapota között. Ezen jellemzők, a speciális felhasználói interfész, és az új OnClickEvent kivételével a komponens listapanelként viselkedik.

Ahogyan a szakasz elején is szerepelt, a Phrases3 példa újabb átdolgozásával keletkező Phrases4 program ennek az új komponensnek a használatát mutatja be. Lényegében az első többszörös-kiválasztó lista helyére egy CheckListBox komponens került True értékű Sorted (rendezett) tulajdonsággal, és az átmásolt elemekkel. Ezután a kódot kellett frissíteni, a ListBox1 CheckListBox1-re

cserélésével és a Selected tömb Checked-el való helyettesítésével. Mindez a ChangeText eljárásban történt, melyből íme egy részlet:

```
for ListItem := 0 to CheckListBox1.Items.Count - 1 do
    if CheckListBox1.Checked [ListItem] then
    begin
      {if the item is selected increase the count}
      Inc (SelectCount);
      if SelectCount = 1 then
      begin
        {store the string of the first selection}
        TmpStr1 := CheckListBox1.Items.Strings
              [ListItem];
        TmpStr2 := TmpStr1;
      end
      else if SelectCount = 2 then ...
```

A 8.21. ábrán az új komponenssel rendelkező program eredménye látható. A komponens fontos tulajdonsága, hogy a felhasználó számára nyilvánvalóbb, hogy az adott listapanel lehetővé teszi a többszörös kiválasztást. A hagyományos lista ellenben ennek semmi jelét nem mutatja.



8.21. ábra *A Phrases4 példa kimenete, az új CheckListBox komponenssel*

Sok lista, kevés hely

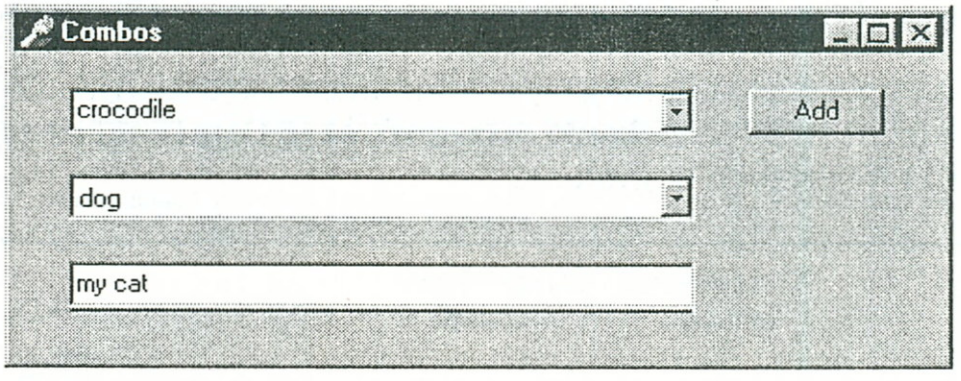
A listapanel sok helyet foglalnak el a képernyőn, és csak egy meghatározott kiválasztást tesznek lehetővé. A felhasználó csak a listában szereplő értékek közül választhat, és nem jelölhet ki olyan lehetőséget, amit a programozó nem látott előre.

A kombinált lista (ComboBox) vezérlő használatával mindkét probléma megoldható. Nagyon hasonlít egy szerkesztő mezőre, és gyakran szöveget is írhatunk bele. Tartalmazza emellett a listák vonásait is, mivel a rajta található nyíl segítségével egy listapanelt jeleníthetünk meg. Még a neve is azt jelzi, hogy két komponens, a szerkesztőmező, és a listapanel kombinációjáról van szó.

A Style (stílus) tulajdonságától függően azonban a kombinált lista viselkedése sokat változhat. Íme az egyes stílusok rövid jellemzése:

- A csDropDown stílus egy jellegzetes kombinált listát definiál, mely közvetlen szerkesztést tesz lehetővé, és kívánságra egy listapanelt jelenít meg.
- A csDropDownList egy szerkesztést nem megengedő kombinált listát határoz meg, amely egy billentyű lenyomásakor kiválasztja a lista adott betűvel kezdődő első szavát.
- A csSimple esetén a listapanel mindig látszik, és a közvetlen szerkesztés is megengedett.
- A csOwnerDrawFixed és csOwnerDrawVariable egy úgynevezett tulajdonos által rajzolt (owner-draw) listán alapszik, mely egyszerű karakterláncok helyett a program által meghatározott grafikus ábrákat tartalmazza.

Az első három típus közötti különbségek megfigyelésére ajánlatos elindítani a hamarosan részletezésre kerülő Combos példát. Amint a 8.22. ábrán, vagy a teszteléskor látható, a Combos három kombinált listát jelenít meg, más-más stílussal.



8.22. ábra *A Combos példa eredménye a kombinált lista három alaptípusával*

A program nagyon egyszerű, a kombinált listák kezdetben ugyanazokat az alapszavakat tartalmazzák – több, mint 20 különböző állat nevét. Az első dobozban található egy Add gomb, mely a beírt szöveget, ha az még nem szerepel benne, hozzáadja a listához. Íme a gomb OnClick eseményéhez kapcsolódó kód:

```
procedure TForm1.ButtonAddClick(Sender: TObject);
begin
with ComboBox1 do
  if (Text <> '') and (Items.IndexOf (Text) < 0) then
    Items.Add (Text);
end;
```

A második kombinált lista segítségével az automatikus kikeresés technikát tapasztalhatjuk ki. Egy billentyű lenyomásával az azzal a betűvel kezdődő első név választható ki, majd a ↑ és ↓ gombokkal tovább mozoghatunk a listában anélkül, hogy kinyitnánk azt. Az iménti technika a kezdőbetűvel és a nyilakkal minden kombinált lista esetén alkalmazható.

A harmadik rész az elsőnek egy változata. Az új elem hozzáadását nem az Add gomb, hanem az ENTER lenyomásakor végezzük el. A kombinált lista OnKeyPress eseményére egy eljárást írunk, ami ellenőrzi, hogy a gomb volt-e, melynek számkódja 13. A többi utasítás a gomb OnClick eseményében lévőkhöz hasonló:

```
procedure TForm1.ComboBox3KeyPress(
  Sender: TObject; var Key: Char);
begin
  {if the user presses the Enter key}
  if Key = Chr (13) then
    with ComboBox3 do
      if (Text <> '') and (Items.IndexOf (Text) < 0)
then
        Items.Add (Text);
end;
```

Új!

A Delphiben megtalálható még egy kombinált listához hasonló felhasználói interfésszel rendelkező összetevő, az új DateTimePicker is, melyet a 11. fejezetben tárgyalunk.

Egy adott határon belüli érték kiválasztása

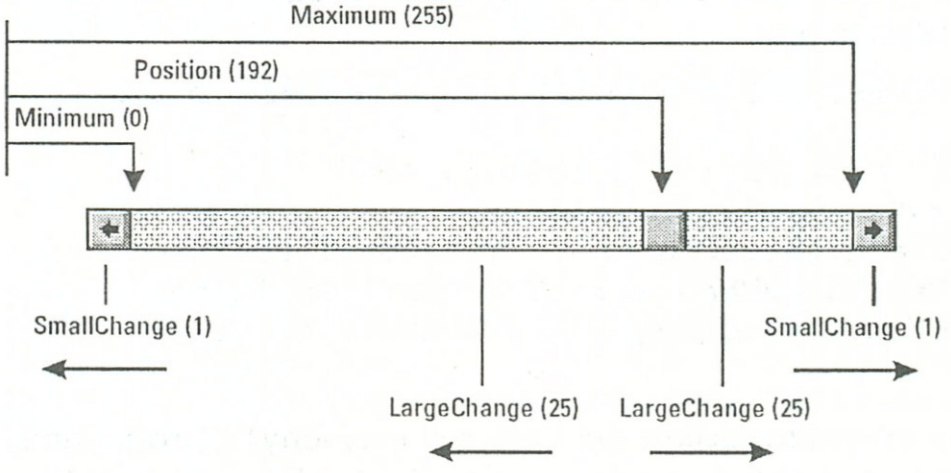
Az utolsó alapvető komponens, amit ebben a fejezetben szeretnék bemutatni a gördítősáv (ScrollBar). Gyakran olyan más komponensekhez kapcsolják, mint például listapanelek és feljegyzés mezők, vagy közvetlenül magához a formhoz csatolják. Ha egy komponenshez gördítősávot rendelünk akkor, az valójában annak a komponensnek lesz a része – tulajdonságainak egyike –, így kevés kapcsolat marad magával a ScrollBar komponenssel. A gördítősávokat tartalmazó formok viszont egyáltalán nem tartalmaznak ScrollBar komponenst, a grafikus elem kijelzésére keretük egy részét használják. A gördítősávokkal felszerelt formokat a 14. fejezetben tárgyaljuk.

A ScrollBar önálló használata elég ritka, főleg, hogy megjelent a Windows 95 TrackBar komponense. Néhány esetben azonban lehet szerepe. Jellegzetes példa számértékek kiválasztása egy kellően nagy intervallumból (a TrackBar-t általában kisebb skálán használják).

A legtöbb Windows programozásról szóló könyv a gördítősáv használatát egy szín kiválasztásának példáján mutatja be, és ez alól mi sem vagyunk kivételek. Aki már találkozott egy ilyen jellegzetes Windows példával, észrevehet egy érdekes dolgot: a Delphi használatával a példa körülbelül negyed annyi idő alatt elkészül, és csak minimális mennyiségű kódot kell írni.

A Scroll Color példa

A ScrollC példa egy egyszerű formpal rendelkezik, három gördítősávval, az azokhoz tartozó címkékkel, egy szintén felcímkézett TrackBar-ral, és néhány komponenssel az aktuális szín megjelenítéséhez. A gördítősávok mindegyike a három alapszín egyikére utal, melyek a Windowsban a vörös, zöld és a kék (RGB – Red, Green, Blue) színek. A címkék a megfelelő szín nevét és az aktuális értéket mutatják.



8.23. ábra *Egy gördítősáv főbb tulajdonságainak grafikus ábrázolása (az értékek a ScrollC példából származnak)*

A gördítősávok jó pár különleges tulajdonságot tartalmaznak. A Min és a Max a lehetséges értékek határait; a Position az aktuális pozíciót; a LargeChange és a SmallChange tulajdonság pedig azt jelzi, hogy a kattintás a sávon, vagy az annak végén lévő nyilakon történt. A tulajdonságok grafikus leírása a példában használt értékekkel a 8.23. ábrán található.

A ScrollC példában minden sáv 0-tól 255-ig terjed. Az intervallum onnan ered, hogy a szín egy DWORD érték, melynek alsó három bájtja képviseli a vörös, zöld, és kék összetevőket (lásd 7. fejezet). A kezdeti pozíció azért 192, mert ha a vörös, zöld, és kék értékek mindegyikét 192-re állítjuk, egy jellegzetes világosszürkét kapunk, amely a formok alapértelmezett színe. A három ScrollBar komponens egyikének szöveges leírása:

```
object ScrollBarRed: TScrollBar
  LargeChange = 25
  Max = 255
  Position = 192
  OnScroll = ScrollBarRedScroll
end
```

A TrackBar komponens hasonló tulajdonságokkal rendelkezik (Min, Max, Position):

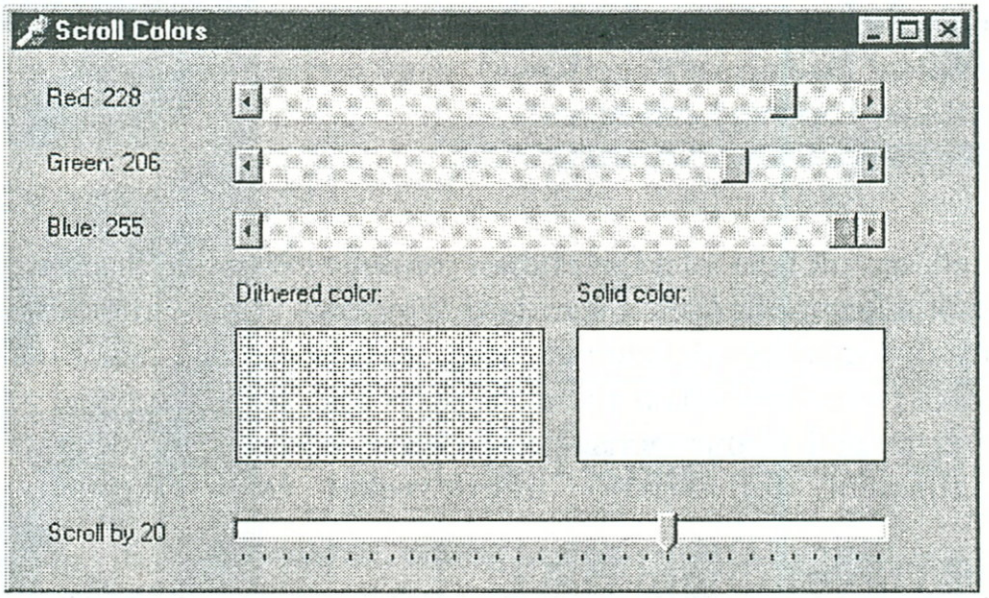
```
object TrackBar1: TTrackBar
  Max = 30
  Min = 1
  Orientation = trHorizontal
  Frequency = 1
  Position = 25
  TickMarks = tmBottomRight
  TickStyle = tsAuto
  OnChange = TrackBar1Change
end
```

A TrackBar-t a három gördítősáv LargeChange tulajdonságának beállítására használjuk az alábbi kód szerint:

```
procedure TFormScroll.TrackBar1Change(Sender: TObject);
begin
  LabelScroll.Caption := 'Scroll by ' +
    IntToStr(TrackBar1.Position);
  ScrollBarGreen.LargeChange := TrackBar1.Position;
  ScrollBarRed.LargeChange := TrackBar1.Position;
  ScrollBarBlue.LargeChange := TrackBar1.Position;
end;
```

Valamelyik gördítősáv megváltozásakor (az OnScroll esemény) a programnak frissítenie kell a megfelelő címkét és a szín megjelenítésére használt formákat. Az első ilyen forma a gördítősávok RGB értékeiből összeálló szín

megjelenítésére szolgál. Ha a forma kitöltésére használt ecset Color tulajdonságába beleírjuk az aktuális színt, egy kevert színt kapunk, mely a videokártya által megjeleníthető alapszínekből áll. A második forma ecsetéhez is ugyanezt a színt rendeltük, így az a kívánt értékhez a lehető legközelebb kerül. Az ecsetek alapbeállításban nem használnak színkeverést, inkább a legközelebbi kirajzolható színt használják. A különbség a 8.24. ábrán és a program futtatásakor látható, bár ez a videokártyától is függ.



8.24. ábra *A ScrollC példa*

A kódban böngészve ráakadhatunk egy harmadik forma komponensre, mely a második forma keretét hivatott utánozni. A valódi keretet egy nagyon széles toll használatával akkorára nagyítjuk, hogy betöltse az egész felületet. Így a toll – a széles keret – színét ténylegesen a forma kitöltésére használjuk. Íme az egyik gördítősávhoz tartozó kód:

```
procedure TFormScroll.ScrollBarRedScroll(Sender:
TObject;
  ScrollCode: TScrollCode; var ScrollPos: Integer);
begin
  LabelRed.Caption := 'Red: ' + IntToStr(ScrollPos);
  Shape1.Brush.Color := RGB (ScrollBarRed.Position,
    ScrollBarGreen.Position, ScrollBarBlue.Position);
  Shape2.Pen.Color := RGB (ScrollBarRed.Position,
    ScrollBarGreen.Position, ScrollBarBlue.Position);
end;
```

Az iménti programrészt minden gördítősávra le kell másolni, a címke és a kiírt szöveg megfelelő helyettesítésével. A második és harmadik utasítás mindig

ugyanaz marad. A színérték meghatározására a Windows RGB függvényét használtuk, mely három 0-255-ig terjedő számot kér, és egy 32 biteset ad vissza a megfelelő színértékkel.

Érdemes megjegyezni, hogy az OnScroll eseménynek három paramétere van: a küldő (sender), az esemény fajtája (ScrollCode) és a jelző végső helye (ScrollPos). Ez a fajta esemény a felhasználó cselekedeteinek szabályozására nagyon pontos vezérlést szolgáltat. A ScrollCode paraméter jelzi, hogy a felhasználó húzza-e a jelző (scTrack, scPosition, vagy scEndScroll), a két nyíl közül valamelyikre kattintott (scLineUp vagy scLineDown), a sávra kattintott valamelyik irányban (scPageUp vagy scPageDown), vagy esetleg megpróbál kijutni az intervallumból (scTop vagy scBottom).

Hogyan tovább?

Ebben a fejezetben elkezdtük feltárni a Delphiben rendelkezésre álló alapvető komponenseket. Ezek a komponensek a standard Windows vezérlőknek és a Windows 95 néhány új vezérlőjének felelnek meg, melyek az alkalmazásokban (az önálló gördítősáv kivételével) roppant népszerűek. Természetesen az alkalmazást fejlettebb Delphi komponensekkel bővítve könnyen építhetünk összetettebb és színesebb felhasználói interfészeket és hatékonyabb programokat.

Az ezt követő fejezetekben (különösen a 11. és 12. fejezetben) részletezünk pár fejlettebb komponenst, viszont a következő két fejezetet két különleges és fontos témának szenteljük: a menük használatának, és a formok egy részletes leírásának. A két mélyreható leírás után, melyek még egér inputról és a közvetlen canvas outputról is szólnak, visszatérünk az komponensek használatához, beleértve a listapanelek grafikus változatait, grideket és még sok mást.

9. FEJEZET

Menük készítése és kezelése

E fejezet tartalma:

- A főmenü felépítése
- Menüparancsok feldolgozása
- Menük módosítása futásidőben
- Grafikus menüpontok
- A rendszermenü testreszabása
- Teljes menü készítése
- Előugró menük

Menük készítése és kezelése

Az eddig elkészített példaprogramokból hiányzott a Windows alkalmazások felhasználói interfészének egyik legfontosabb eleme, a menü. Igaz ugyan, hogy minden formunknak volt rendszermenüje, de ennek felhasználhatósága nagyon korlátozott volt. Valódi alkalmazásokban azonban a menü a programfejlesztés egyik központi témája, hiszen, bár a felhasználó kattintgathat, vagy néha még az egérrel el is húzhat dolgokat bizonyos opciók kiválasztásához, a legösszetettebb feladatok elvégzéséhez általában menükre van szükség. Gondoljunk bármely általunk használt alkalmazásra, és a használat során kiadott menü parancsok számára, ideértve azokat is, amiket gyorsbillentyűvel helyettesítettünk, mint például a CTRL+C az **Edit** (Szerkesztés) menü **Copy** (Másol) parancsa helyett.

A menük annyira fontosak, hogy szinte minden valódi Windows alkalmazásnak van legalább egy. A valóságban egy alkalmazásnak lehet több futás alatt változó menüje – erről még lesz szó később –, több helyi menüje – ezeket általában a jobb egérgomb segítségével hozhatjuk elő –, és akár más, saját, testre szabott rendszer menüje is.

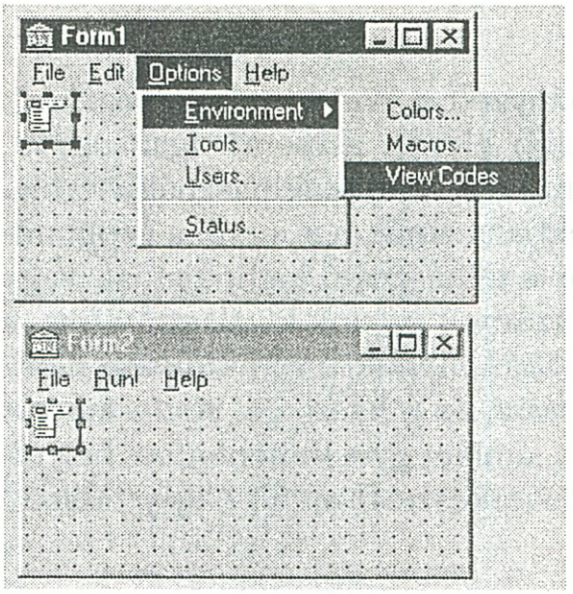
A Delphit készítő programozók a menüket annyira fontosnak tartották, hogy a megfelelő komponenseket a komponens paletta Standard oldalára tették.

A főmenü felépítése

Mielőtt megnéznénk Delphiben a menüket, foglaljunk össze néhány általános tudnivalót a menükről és felépítésükről. A menük általában két szintből állnak. A menüsor az ablak címsora alatt található, és a legördülő menük neveit sorolja fel, amik mindegyike több pontból állhat. A menü struktúra azonban nagyon rugalmas, így lehetőségünk van arra, hogy a menüsorba parancsokat helyezzünk, vagy, hogy egy legördülő menübe egy másik legördülő menüt tegyünk.

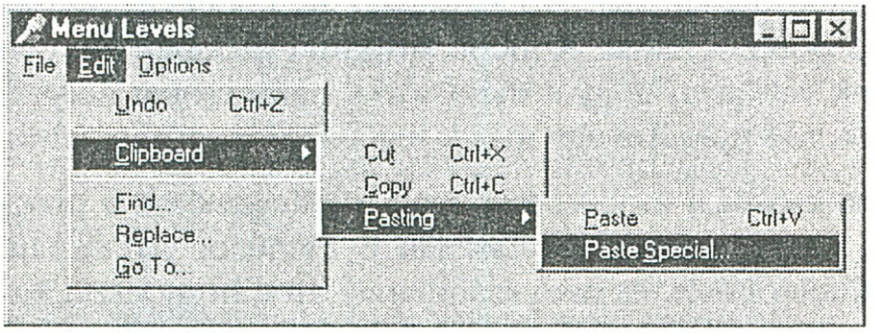
Ha lehet, ne tegyünk parancsokat a menüsorba, mert a felhasználók a programmal való ismerkedés során gyakran kiválasztják a menüsor parancsait, hogy megnézzék a menük felépítését, és nem számítanak arra, hogy ezzel parancsot is kiadhatnak. Ha valami miatt mégis parancsot kell ide tennünk, akkor legalább tegyünk utána egy felkiáltójelet, ahogy az a 9.1. ábrán látható. A felkiáltójel használata bevett jelölés, de a legtöbb felhasználó még soha nem találkozott ezzel a szokással, ezért a legjobb, ha ezt elkerüljük azzal, hogy készítünk egy egyetlen menüpontot tartalmazó legördülő menüt. Erre jellemző példa az a Súgó menü, amiben csak egyetlen Névjegy parancs található.

A legördülő menükből nyíló legördülő menük már sokkal gyakoribbak, és erre az esetre a Windowsnak van is egy jelölése, egy kis háromszög a menü jobb szélén (9.1. ábra). Azt viszont vegyük mindig figyelembe, hogy a második, vagy későbbi szinteken lévő legördülő menük elérése több időt vesz igénybe, és fárasztóvá válhat ezek használata.



9.1. ábra *A tipikus kétszintű menü struktúra mellett legördülő menüt tehetünk másik legördülő menübe is (fent) és menü parancs lehet a menüsoron is (lent)*

Sokszor, alsóbb szintű legördülő menük helyett elég az is, ha elválasztó vonalak segítségével csoportosítjuk a parancsokat. Kiterjedt, többszintű menüre példa a 9.2. ábra, amit ki is próbálhatunk a CD-n található Levels (szintek) példa futtatásával. Mivel ez annak a bemutatása, hogy programozáskor mit kerüljünk, ezért a menü leírását nem adjuk meg (a csoportokba rendezett kapcsolódó menüpontokra látható példa a 9.3. ábrán).



9.2. ábra *A Levels példa többszintű menüje*

A menüpontok különböző szerepei

Most nézzük meg a menüpontokat, a menüstruktúrában elfoglalt helyüktől függetlenül. A menüpontok három alapvető csoportba sorolhatók:

- A parancsok azok a menüpontok, amik valamilyen funkciót indítanak el. Nincsen különleges jelölésük.

- Állapot állítók azok a menüpontok, amikkel egy opciót kapcsolhatunk ki vagy be, vagy egy elem állapotát változtathatjuk meg. Ezeknél a pontoknak a bal oldalán általában egy pipa jelzi, ha aktívak. Ilyenkor a pont kiválasztása az ellenkező hatást eredményezi.
- Dialógus menüpontoknak azokat a pontokat nevezzük, amiknek hatására egy dialógusablak jelenik meg. A valódi különbség ezek, és a többi menüpont között az, hogy a felhasználó megnézheti a kapcsolódó menüpont lehetséges hatásait, és ha meggondolta magát, a Mégse gombbal leállíthatja a végrehajtást. Ezeket a parancsokat célszerű vizuális jelöléssel is ellátni, ami a szöveg után elhelyezett három pont szokott lenni.

Tipp

Windows 95-ben a megszokott pipa állapotjelző mellett használhatunk rádiógombokat is, amiknél egy pont a jelölés. Ezek ugyanolyan választási lehetőségeket jeleznek, mint a RadioButton (rádió gomb) komponens, vagyis ha az egyiket kiválasztjuk, akkor az automatikusan kikapcsolja a csoport többi tagját. A rádió menüvel egy későbbi példában fogunk foglalkozni.

Menü készítése a Menü Tervezővel

A Delphi rendelkezik egy eszközzel a menük szerkesztésére, ez a Menü Tervező (Menu Designer). Ezt az eszközt úgy érhetjük el, ha egy formra felteszünk egy MainMenu (főmenü) komponenst, és duplán kattintunk rajta. Ne törődjünk azzal, hogy hova tesszük a komponenst a formon, hiszen ettől függetlenül a menü mindig helyesen, a form címsora alatt jelenik meg.

Megjegyzés

Hogy pontosak legyünk, a form a címsora alatt a Menu tulajdonságában beállított menüt jeleníti meg, ami automatikusan beállításra kerül a form első főmenü komponensének elkészítésekor. Ha a formnak több mint egy főmenü komponense van, akkor ezt a tulajdonságot célszerű nekünk beállítani, de mind tervezés, mind futás alatt megváltoztathatjuk.

A Menü Tervező nagyon sokoldalú: lehetővé teszi, hogy menüket készítsünk egyszerűen, a parancsok neveinek leírásával, hogy húzással mozgassunk menüpontokat, vagy almenüket, és hogy az egyes pontok tulajdonságait egyszerűen beállíthassuk. Még azt is megtehetjük vele, hogy a menüsorba parancsokat helyezünk – ez történik mindig, amikor nem írunk egyetlen elemet sem a kapcsolódó legördülő menü részbe –, vagy készíthetünk alsóbb szintű legördülő menüket is a Menü Tervező gyorsmenüjének – ezt a jobb egér gombbal érhetjük el – **Create Submenu** (almenü készítése) parancsával.

A Menü Tervező másik nagyon fontos szolgáltatása, hogy készíthetünk menüket sablonokból. Saját sablonokat is könnyen létrehozhatunk. Ehhez készítsünk egy menüt, és a gyorsmenü **Save As Template** (mentés sablonként) parancsával adjuk a listához. Ennek akkor lehet értelme, ha ugyanazt a menüt két alkalmazásban, vagy egy alkalmazás két különböző formján kell felhasználnunk.

A menü általános felépítése

Ha már régebb óta használunk Windows alkalmazásokat, akkor észrevehettük, hogy az alkalmazás menüstruktúrája nem csupán az azt készítő programozók ötleteitől függ. A Windowsban több konvenció létezik a parancsok menükbe rendezésére. Ezeket a szabályokat felismerhetjük, ha megnézünk néhány népszerűbb alkalmazást.

Az alkalmazás menüsora általában a Fájl (File) menüvel kezdődik, ezt követi a Szerkesztés (Edit), Nézet (View), majd ezek után jönnek az alkalmazásra jellemző parancsok. A sor végén van a Beállítások (Options) és az Ablak (Window) – ez utóbbi csak MDI, más szóval több dokumentumot kezelő alkalmazásoknál –, és mindig a Súgó (Help) az utolsó. Mindegyik legördülő menünek megvan a megszokott felépítése, de a ténylegesen megjelenő elemek száma az alkalmazástól függ. Például a Fájl menüben általában a következő parancsok vannak: Új (New), Megnyitás (Open), Mentés (Save), Mentés másként (Save As), Nyomtatás (Print), Oldalbeállítás (Print Setup), és Kilépés (Exit).

Gyorsbillentyűk, forróbillentyűk

A menüpontok neveiben általában szerepel egy aláhúzott betű, egy úgynevezett forróbillentyű. Ez a betű, ami általában a menüpont nevének kezdőbetűje, arra szolgál, hogy a menüpontot billentyűzetről kiválaszthassuk. Az Alt mellett lenyomott aláhúzott betű a megfelelő legördülő menüt választja ki. A menüben szereplő másik aláhúzott betű lenyomásával egy parancsot adhatunk ki.

Természetesen a menüsor mindegyik elemének más aláhúzott betűvel kell rendelkeznie, és ez igaz egy adott legördülő menün belül is; különböző menükben persze lehet ugyanaz a betű aláhúzott. Az aláhúzott betű jelöléséhez egyszerűen tegyünk egy „&" jelet a karakter elé, mint például a Menté&s

másként (Save &As) vagy &Fájl. Ezekben a példákban az aláhúzott betű a Mentés másként esetében az s, a Fájlnál pedig az F.

A menük esetében van még egy másik lehetőség is, a gyorsbillentyűk alkalmazása. Ha egy menüpont mellett egy billentyű, vagy billentyű kombináció látható, akkor az azt jelenti, hogy az adott billentyűk megnyomásával kiadhatjuk azt a parancsot. Bár a parancsok kiadása egérrel könnyebb, de lassabb is, különösen billentyűzetet használó alkalmazásoknál, hiszen a billentyűzetről át kell nyúlnunk az egérhez. Az Alt és aláhúzott betűk használata már valamivel gyorsabb, de még mindig legalább két gombnyomást igényel. A gyorsbillentyűk általában egy speciális billentyű és egy másik gomb egyidejű lenyomásából állnak, mint például a Ctrl+C. A Windows meg sem jeleníti a kapcsolódó legördülő menüt, ezért a parancs végrehajtása is gyorsabb.

Delphiben a gyorsbillentyűk menüpontokhoz kapcsolása nagyon egyszerű, csak állítsunk be egy értéket a ShortCut (Gyorsbillentyű) tulajdonságnál, ahol a megszokott kombinációk közül választhatunk: Ctrl vagy Shift és szinte bármely másik gomb.

A programhoz adhatunk gyorsbillentyűket anélkül is, hogy a programnak menüje lenne. Például készíthetünk előugró menüt, formhoz kapcsolhatjuk, a form PopupMenu (előugró menü) tulajdonságának beállításával, az összes menüpont Visible (látható) tulajdonságát hamisra (False) állíthatjuk; a felhasználó soha nem fogja látni a menüt, de – a Súgónkban dokumentált – összes gyorsbillentyű működni fog. Ha ez nem világos, akkor a nézzük meg a CD-n található HShort példát.

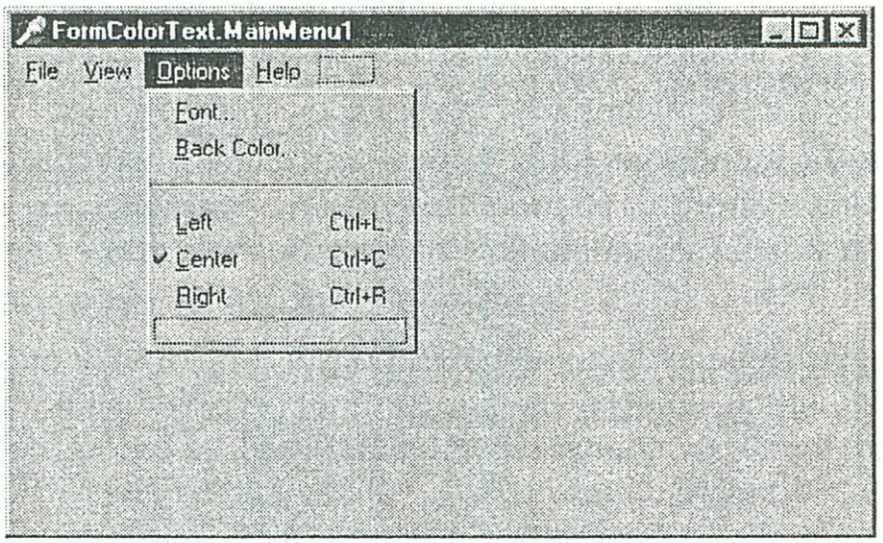
Előre definiált menüsablonok használata

Alkalmazásunk általános irányelveket követő menüjének fejlesztéséhez a Delphi több előre definiált menüsablont kínál. A sablonok között megtalálható két különböző File legördülő menü, egy Edit menü, amiben OLE parancsok is vannak, egy Window menü, és kétféle Help menü. Van még teljes MDI menüsor sablon is, amiben megtalálható ugyanez a négy kategória.

Ezeknek a menü sablonoknak a használata több előnnyel jár. Először is gyorsabb már létező menüket felhasználni, mint elölről felépíteni egyet. Másodszor a menü sablonok a Windows iránymutatásait követik a parancsok elnevezésekor, a gyorsbillentyűk kiosztásakor, és így tovább. Természetesen ezeknek a menüknek csak állományokon dolgozó alkalmazásoknál van értelmük, hiszen ha a programunk nem kezel állományokat, nem tud szerkeszteni, és nem MDI típusú, akkor legfeljebb a Help sablont tudjuk felhasználni.

Menüparancsok feldolgozása

Első menüs példánk, a MenuOne elkészítéséhez az előző fejezetbeli LabelCo példát fogjuk továbbfejleszteni, aminek az eredeti formját a 8.3. ábrán láthattuk. Az új változatban lesz egy MainMenu komponens, ennek négy legördülő menüje lesz: a File egyetlen Exit menüponttal, a View egyetlen Toolbar (eszköztár) elemmel, az Options (beállítások) menü több opcióval, valamint a Help (súgó) menü az About (névjegy) ponttal. Ezt a menüsort láthatjuk a Menü Tervezőben a 9.3. ábrán.



9.3. ábra *A MenuOne példa menüsora*

Legördülő menükben elválasztót helyezhetünk el, ha a parancs szövegének egy mínusz jelet adunk meg. Ne használjuk a Break (törés) tulajdonságot, mert egészen ritka esetektől eltekintve ez elronthatja a menünket, legjobb, ha el is felejtjük, hogy ilyen tulajdonság létezik.

Tipp

Természetesen a Break tulajdonságnak is megvan a haszna, különben nem lenne a komponens része. Viszont sokkal érthetőbb, ha a különböző lehetséges értékeit más névvel illetjük: NewLine (új sor), vagy NewColumn (új oszlop). Ha a menüsor egyik elemének mbMenuBarBreak (MenüSorTörés) (vagy NewLine) értéke van, akkor ez az elem egy új, következő sorban jelenik meg. Ha egy menüpontnak mbMenuBreak (menütörés) (vagy NewColumn) értéke van, akkor ez a legördülő menü egy új, következő oszlopában jelenik meg. Ezek nem túl gyakran használt tulajdonságok.

A Menü Tervező által generált kód

Miután elkészítettük a menüt, nézzük meg az Object Inspector által mutatott komponensek listáját, vagy nyissuk ki a form szöveges leírását tartalmazó DFM állományt, amiben megtalálhatjuk a menü szöveges leírását is. Itt következnek a form szöveges leírásának a menüvel kapcsolatos részei:

```
object MainMenu1: TMainMenu
    object File1: TMenuItem
      Caption = '&File'
      object Exit1: TMenuItem
        Caption = 'E&xit'
        OnClick = Exit1Click
      end
    end
    object View1: TMenuItem
      Caption = '&View'
      object Toolbar1: TMenuItem
        Caption = '&Toolbar'
        Checked = True
        OnClick = Toolbar1Click
      end
    end
    object Options1: TMenuItem
      Caption = '&Options'
      object Font1: TMenuItem
        Caption = '&Font...'
        OnClick = Font1Click
      end
      object BackColor1: TMenuItem
        Caption = '&Back Color...'
        OnClick = BackColor1Click
      end
      object N1: TMenuItem
        Caption = '-'
      end
      object Left1: TMenuItem
        Caption = '&Left'
        ShortCut = 16460 // stands for Ctrl+L
        OnClick = Left1Click
      end
      object Center1: TMenuItem
        Caption = '&Center'
        Checked = True
        ShortCut = 16451 // stands for Ctrl+C
        OnClick = Center1Click
      end
      object Right1: TMenuItem
```

```
        Caption = '&Right'
        ShortCut = 16466 // stands for Ctrl+R
        OnClick = Right1Click
      end
    end
    object Help1: TMenuItem
      Caption = '&Help'
      object About1: TMenuItem
        Caption = '&About Menu One...'
        OnClick = About1Click
      end
    end
  end
```

Amint az a fenti listából látható, minden menüponthoz külön komponens tartozik, és meglepő, de még az elválasztókhoz is. Ezeknek a komponenseknek a neveit a Delphi magától generálja a menüpont nevéből a következő szabályok alapján:

- Minden üres és különleges karaktert eltávolít.
- Ha nem marad egyetlen karakter sem, akkor a név egy N betű lesz.
- A végéhez hozzátesz egy számot, 1-et, ha ez az első ilyen nevű menüpont, nagyobbat, ha nem.

Az összes ilyen új komponens látható az Object Inspectorban, és közvetlenül kiválaszthatjuk őket, vagy mozoghatunk közöttük a Menü Tervező megnyitása után. Az elemek listája szerepel még a form osztálydefiníciójánál a komponensek között:

```
type
  TFormColorText = class(TForm)
    MainMenu1: TMainMenu;
    Options1: TMenuItem;
    Font1: TMenuItem;
    BackColor1: TMenuItem;
    N1: TMenuItem;
    Left1: TMenuItem;
    Center1: TMenuItem;
    Right1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    File1: TMenuItem;
    Exit1: TMenuItem;
    View1: TMenuItem;
    Toolbar1: TMenuItem;
```

Tipp

Olyan menüpontok meződeklarációit, amikre nem hivatkozunk programból – például az elválasztók –, eltávolíthatjuk, akár az N1-et fentebb. Ezek létrejönnek, mert szerepelnek a Form leíró állományban, de a forrásból nem tudunk rájuk hivatkozni. Mivel ezek mindenképpen létrejönnek, nem nyerünk túl sok memóriát, csak a form osztálybeli referencia helyét, de a felesleges sorok eltávolítása megnövelheti a kód olvashatóságát.

Hogy egy menüpontra válaszolni tudjuk, a menüpont OnClick eseményéhez kell definiálnunk egy metódust. A legördülő menük OnClick eseményét csak különleges esetekben használjuk, például olyankor, ha meg akarjuk nézni, hogy az alatta lévő pontokat érvényteleníteni kell-e. Az elválasztók OnClick eseménye teljesen fölösleges, mert ez sohasem aktiválódik.

Tipp

Miután elkészültünk a form főmenüjével, és az megjelenik a címsor alatt, az OnClick eseménykezelő metódusokat egyszerűen a parancs kiválasztásával elkészíthetjük. Ha már létezik ilyen kezelő, akkor a Delphi a megfelelő kódrészletet kínálja fel, egyébként új metódust ad a formhoz.

A MenuOne példa forrása

A MenuOne példa kódja nagyon egyszerű, és hasonló a LabelCo példáéhoz, amit kibővít. A háttérszínre vonatkozó menüparancs OnClick eseménye, és az eszköztár megfelelő gombja ugyanarra a metódusra mutat:

```
object BtnBackColor: TButton
  Caption = '&Back Color...'
  OnClick = BackColor1Click
end
```

Ennek a metódusnak a forrása megegyezik az előző verzióéval.

A betűtípussal kapcsolatos parancs és gomb egyaránt a Font1Click metódushoz kötődik, ami most a színválasztó dialógus helyett a fontválasztót kínálja fel. Ennek a kódja tömör a „with" kifejezés használata miatt:

```
procedure TFormColorText.Font1Click(Sender: TObject);
begin
  with FontDialog1 do
  begin
    Font := Label1.Font;
    if Execute then
      Label1.Font := Font;
  end;
end;
```

Az Option menü másik három menüpontjának – és az eszköztár utolsó három gombjának – alapvetően ugyanaz a kódja, mint a LabelCo példában: az OnClick eseménykezelőik beállítják a címke Alignment tulajdonságát. Ez jól működik, csak nem követi a felhasználói felülettervezés alapelveit. Ha egy menüben több, összefüggő választási lehetőségünk van, akkor a kiválasztott mellett célszerű egy pipát elhelyezni.

Ehhez két különböző dolgot kell tennünk. Először el kell helyeznünk a pipát az alapértelmezett beállítás, a Center, mellett azzal, hogy az Object Inspectorban a menüpont Checked (kipipált) tulajdonságát igazra állítjuk. Másodszor módosítanunk kell a kódot, hogy a változásokkor a pipa jó helyen legyen, ahogy azt a 9.4. ábrán láthatjuk:

```
procedure TFormColorText.Left1Click(Sender: TObject);
begin
  Label1.Alignment := taLeftJustify;
  Left1.Checked := True;
  Center1.Checked := False;
  Right1.Checked := False;
end;
```

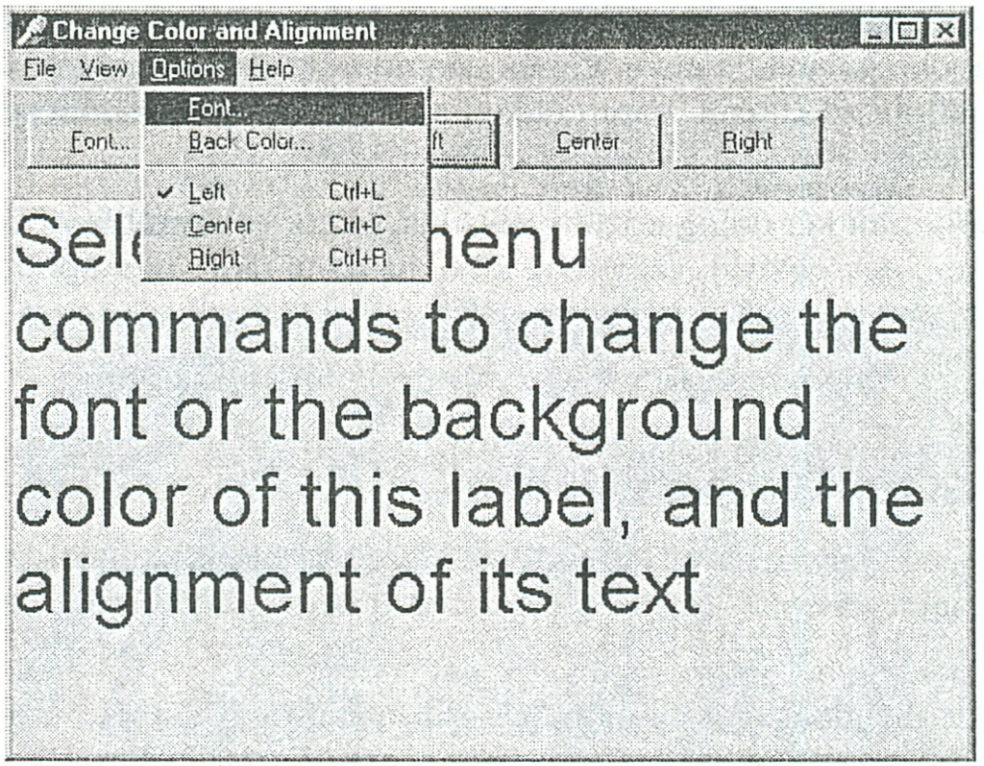
A másik két metódus ugyanilyen, elég, ha az utolsó három értékadást lemásoljuk, elhelyezzük a bennük, és a három Checked tulajdonság értékét úgy módosítjuk, hogy mindig a megfelelő egy legyen igaz. A pipák eltávolítására van hatékonyabb módszer is, de az igazi megoldást a rádiógomb formátumú menüpontok használata jelentené. Ezt a technikát később nézzük meg.

A View menü Toolbar parancsa tipikus példa a pipával jelölésre, ami azt mutathatja, hogy az eszköztár látható-e, vagy sem. A kód a következő:

```
procedure TFormColorText.Toolbar1Click(Sender: TObject);
begin
  Panel1.Visible := not Panel1.Visible;
  Toolbar1.Checked := Panel1.Visible;
end;
```

Ez a részlet a panel Visible tulajdonságát állítja át, és a pipát – a menüpont Checked tulajdonságát – ennek megfelelően kapcsolja. Arra figyeljünk, hogy a tulajdonság kezdeti értéke tükrözze a panel kezdeti állapotát. A Help menü

About, és a File menü Exit pontjai csak egy üzenet ablakot jelenítenek meg, vagy a Form Close metódusát hívják, és annyira egyszerűek, hogy a forráskódjuk bemutatásától eltekintünk. Akit érdekel, az megtalálhatja a CD-n az MenuOneF.Pas állományban.



9.4. ábra *A MenuOne példa legördülő menüjében a jel az aktuálisan kiválasztottat mutatja*

Menük módosítása futásidőben

Windowsban a menük futás közbeni megváltoztatására több lehetőségünk is adódik. Először az egyetlen menüpontra vonatkozó változtatásokat nézzük meg, azután átváltunk a legördülő menükre, készítünk egy rugalmas főmenüt, és megváltoztatjuk a pipa képét.

Menüpontok változtatása futásidőben

Fontos tudni, hogy a menüpontok megváltozhatnak futásidőben. Például a nem kiválasztható menüparancsok elszürkíthetők, így a felhasználó semmiképp nem tudja kiválasztani azokat. Általában nem szabad elrejtenünk a nem kiválasztható parancsokat, elég, ha kikapcsolttá tesszük őket. Ez a bevett technika arra, hogy a felhasználóval tudassuk, hogy a parancs nem elérhető, máskülönben azt hihetné, hogy a parancs valahol máshol van a menüstruktúrában, és keresni kezdené.

Másik vizuális változtatás a pipa használata, amit az alkalmazás könnyedén ki és bekapcsolhat, amint azt az előző példában láthattuk. Néha, állapot állító menüpontoknál, a könnyebb használhatóság érdekében a menüpont szövegét is kicserélhetjük. Erre példa lehet az Eszköztár megjelenítése parancs. Ha ezt kiválasztjuk, akkor egy pipa jelenik meg a név mellett, és az eszköztár megjelenik; ha ezek után újra kiválasztjuk ezt a pontot, az eszköztár eltűnik. A parancs pont az ellenkezőjét okozta, mint a neve. Ennek elkerülésére használhatunk a menüpontnál két különböző címkét, mint például az Eszköztár Megjelenítése és az Eszköztár Elrejtése.

Ha a felhasználó több mint két dolog közül választhat, akkor célszerűbb több menüpontot használni, egy pipával, vagy még jobb, ha az új rádió menüpont interfészt használjuk, ami lassan szabványossá válik.

Általában három tulajdonságot használunk egy menüpont módosítására:

- A Checked tulajdonságot használtuk a fenti példában, hogy megjelenítsük a pipát a menüpont mellett.
- Az Enabled (engedélyezett) tulajdonság segítségével elszürkíthetjük a menüpontot, amit innentől kezdve a felhasználó nem választhat ki, de látható marad.
- Az utolsó ilyen tulajdonság a Caption, a menüpont szövege, amit megváltoztathatunk, hogy tükrözze a parancs mostani hatását, ahogy azt fentebb láthattuk.

Ezeknek a tulajdonságoknak a használatát mutatjuk be oly módon, hogy a MenuOne példát új menüpontokkal bővítjük. A projekt új neve MenuOne2. Két új menüpont került a View menübe – a Hide Label (címke elrejtése), és a Fixed Font (rögzített font) –, és az Options menübe is került két új pont – a Fixed View (rögzített nézet), és a Disable Help (súgó kikapcsolása) –, valamint megjelent néhány új elválasztó is.

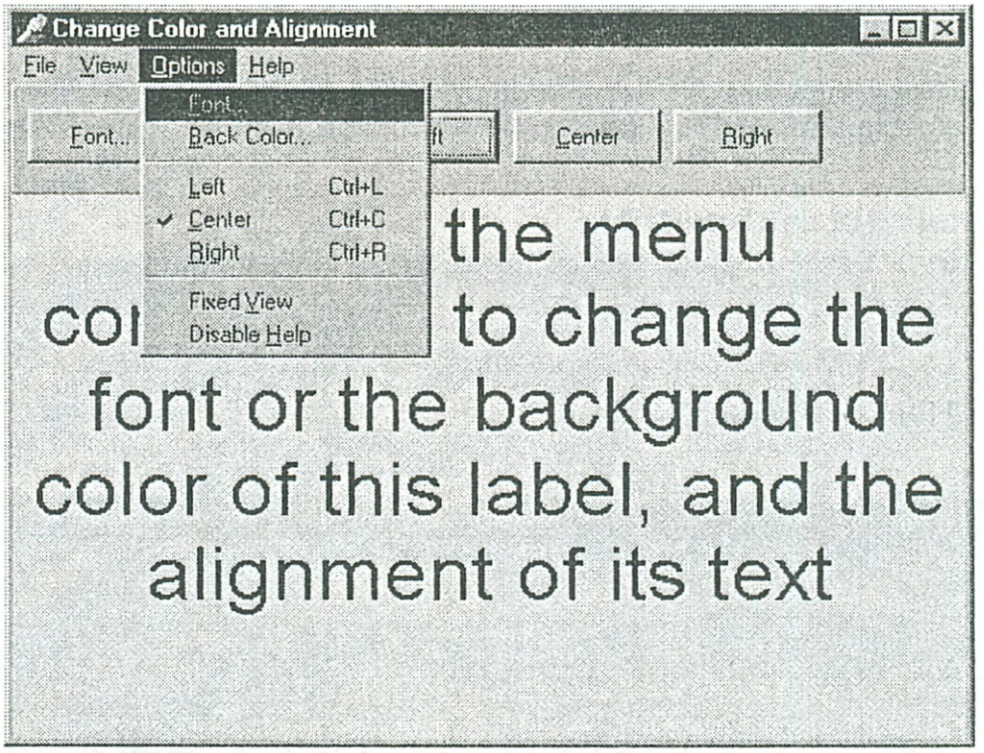
A Hide Label menüpont bemutatja a program állapotától függő menüpont nevek használatát. Ez az OnClick eseménykezelője:

```
procedure TFormColorText.HideLabel1Click(Sender:
TObject);
begin
  Label1.Visible := not Label1.Visible;
  if Label1.Visible then
    HideLabel1.Caption := 'Hide &Label'
  else
    HideLabel1.Caption := 'Show &Label'
end;
```

Menüpontok kikapcsolása és legördülő menük elrejtése

A MenuOne2 másik három új menüpontja arra szolgál, hogy más menüpontokat, vagy legördülő menüket elrejtsünk. A View menü Fixed Font parancsa az Options menü Font pontját kapcsolja ki, ahogy az a 9.5 ábrán látható:

```
procedure TFormColorText.FixedFont1Click(Sender:
TObject);
begin
  ToggleCheck (FixedFont1);
  Font1.Enabled := not Font1.Enabled;
end;
```



9.5. ábra *A kikapcsolt menüpont még ott van, de nem kiválasztható. Ez a kép a MenuOne2 példából van*

Ez a metódus meghívja a ToggleCheck (pipa kapcsolás) eljárást, amit azért írtunk, hogy a pipa jelölés átkapcsolását leegyszerűsítsük, hiszen mind a három menüpontunk ugyanezt teszi. Íme a forrása:

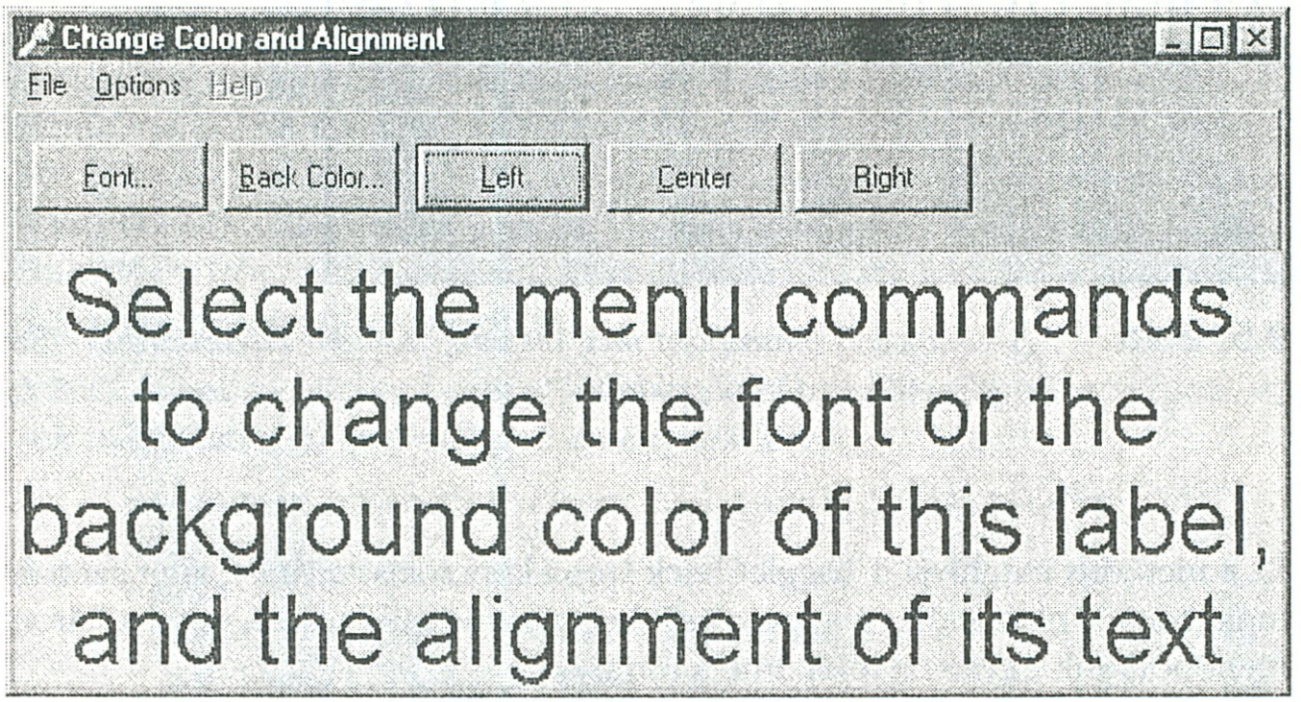
```
procedure ToggleCheck (Item: TMenuItem);
begin
  Item.Checked := not Item.Checked;
end;
```

Már beszéltünk arról, hogy nem tanácsos menüpontokat elrejteni a Visible (látható) tulajdonságuk kikapcsolásával, mert a felhasználó ezek után más

legördülő menükben kezdi keresni azokat. A menüpontokat célszerű csak kikapcsolni, ha nem akarjuk elérhetővé tenni. Egész legördülő menük elrejtése viszont megszokott dolog; sok alkalmazásban a látható legördülő menük az alkalmazás ablakának állapotához igazodnak. Bár technikailag lehetséges a legördülő menük kikapcsolása is, de ez nem szokás. A hátralévő két menüpont ezt a két műveletet végzi, amint az a megfelelő OnClick eseménykezelőből is látható:

```
procedure TFormColorText.FixedView1Click(Sender:
TObject);
begin
  ToggleCheck (FixedView1);
  View1.Visible := not View1.Visible;
end;

procedure TFormColorText.DisableHelp1Click(Sender:
TObject);
begin
  ToggleCheck (DisableHelp1);
  Help1.Enabled := not Help1.Enabled;
end;
```

A 9.6. ábrán ennek a két műveletnek az eredményét láthatjuk. A View legördülő menü eltűnt, a Help pedig elszürkült.

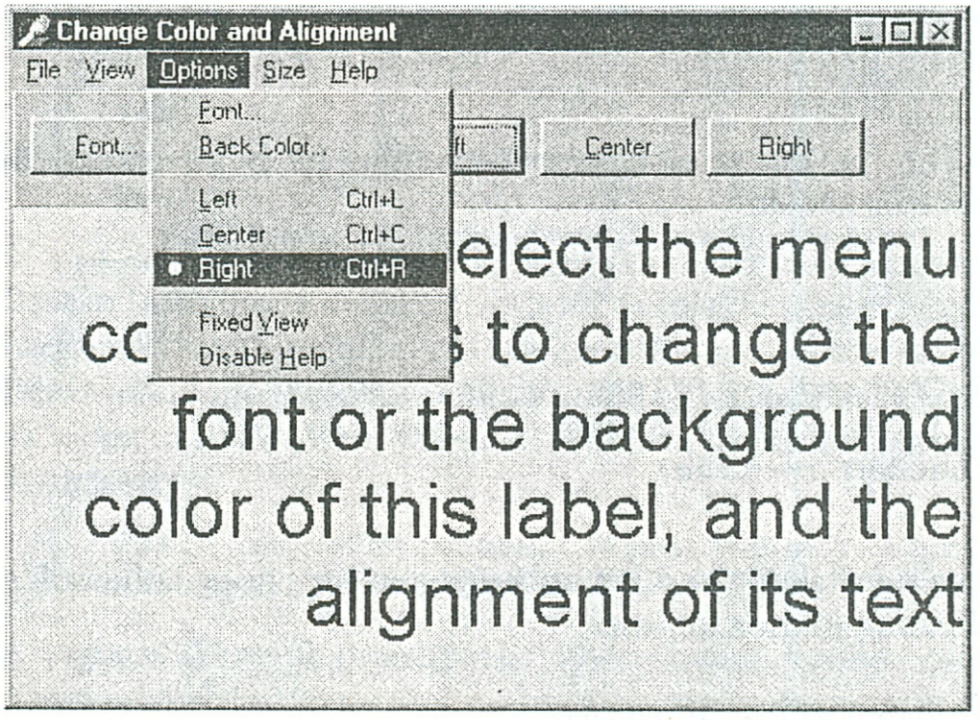


9.6. ábra *A MenuOne2 példa formja futásidőben, amikor az egyik legördülő menü kikapcsolt, a másik pedig rejtett*

Rádió menüpontok használata

A pipák használata mellett Windows 95-ben és Windows NT 4.0-ban lehetőség nyílik rádió menüpontok használatára. Ezek nem csak másfajta felhasználói felületet nyújtanak, de másként is viselkednek, emiatt egyszerűbb kódot eredményeznek, mert a rendszer néhány dolgot megcsinál helyettünk. Erre az új elemre példa a Windows Intéző Nézet menüje.

Delphiben egyszerűen igazra állítjuk a MenuItem komponens RadioItem (rádió menüpont) tulajdonságát, és máris egy új jelölés jelenik meg a pont mellett. Ha ezt a tulajdonságot több egymás utáni menüpontra is beállítjuk, és a GroupIndex (csoportindex) tulajdonságukat ugyanolyan értékre módosítjuk, akkor ezek rádió gombokként fognak működni, és az új jelölést használják, ahogy az a 9.7. ábrán is látható. Ez azt jelenti, hogy a csoportnak egyszerre csak egyetlen eleme lehet kiválasztott, most azonban a MenuOne első változatától eltérően nem nekünk kell az összes többit lekapcsolni, hanem ez magától történik.



9.7. ábra *A MenuOne3 példa új rádió menüpontjainak jelölése*

Itt következik az, hogy hogyan tudjuk megvalósítani ezt a MenuOne példa harmadik változatában, aminek még vannak más változásai is, amikről a következő részben lesz szó. A lenti lista a csoportba foglalt menüpontok megváltozott szöveges leírását mutatja.

```
object Options1: TMenuItem
  Caption = '&Options'
  ...
  object Left1: TMenuItem
    Caption = '&Left'
    GroupIndex = 1
    RadioItem = True
    OnClick = Left1Click
  end
  object Center1: TMenuItem
    Caption = '&Center'
    Checked = True
    GroupIndex = 1
    RadioItem = True
    OnClick = Center1Click
  end
  object Right1: TMenuItem
    Caption = '&Right'
    GroupIndex = 1
    RadioItem = True
    OnClick = Right1Click
  end
```

Ez a kód működik így is, de le is egyszerűsíthetjük, ha a következő új Right1Click metódust használjuk:

```
procedure TFormColorText.Right1Click(Sender: TObject);
begin
  Label1.Alignment := taRightJustify;
  // Left1.Checked := False; // now useless
  // Center1.Checked := False; // now useless
  Right1.Checked := True;
end;
```

Egyszerűen megjegyzéssé alakítottuk a két szükségtelen sort, hogy láthassuk a különbséget a régebbi verziókhoz képest.

Menüpontok készítése dinamikusan

A menüpontok és legördülő menük eddigi futásidejű változtatásai mind tulajdonságok közvetlen módosításán alapultak. Ezeknek a komponenseknek azonban van néhány érdekes metódusa, mint az Insert (beszúrás) és Remove (eltávolítás), amikkel további változtatásokat is megtehetünk.

A TMenuItem osztály – amit a Delphi a menüpontok és a legördülő menükre is egyaránt használ – alapötlete az, hogy mindegyik objektum menüpontok listáját tartalmazza. Minden ilyen pontnak ugyanaz a szerkezete, egyfajta rekurzív módon. A legördülő menük almenük sorozatából állnak, amik mindegyikének lehetnek almenüi, amik mind tartalmazhatnak saját almenüket, és így tovább.

Létező menü struktúrák feltérképezéséhez használhatjuk az Items (pontok) tulajdonságot, ami a menüpontok listáját tartalmazza, valamint a Count (számosság) tulajdonságot is, ami az almenüpontok számát tartalmazza. Új menüpontok, vagy akár egész új legördülő menük hozzáadása nagyon egyszerű, ennél némiképp bonyolultabb az új menüpontok hatásának lekezelése. Ehhez, a Delphi környezet különleges szolgáltatásai nélkül írnunk kell egy üzenetkezelő metódust, amit azután az új ponthoz rendelünk az OnClick tulajdonságának átállításával. Másik megoldásként használhatjuk ugyanazt a metódust több OnClick eseménynél is, és a Sender (küldő) paraméterrel eldönthetjük, hogy melyik parancsot kell végrehajtanunk.

Mindezek a lehetőségek megtalálhatók a MenuOne3 példában. Amint elindítjuk a programot, egy új, a font méretét állító legördülő menü jelenik meg a formon. Ahelyett, hogy készítenénk egy csomó menüpontot, amik a 8 és 48 közötti méreteket jelölik, ezt az ismétlődő munkát a programra hagyjuk. Bár a legördülő részt elkészíthettük volna tervezési időben, és utána csak a menüpontokat kellett volna dinamikusan hozzáadni, de jobbnak láttuk, ha a teljes példát megmutatjuk, hogy később ilyen esetekben ezt felhasználhassuk.

Új menüpont, vagy legördülő menü készítéséhez egyszerűen meg kell hívni a TMenuItem Create konstruktorát:

```
var
  PullDown: TMenuItem;
begin
  PullDown := TMenuItem.Create (self);
```

Ezután beállítjuk a címkéjét, és a többi tulajdonságát, végül pedig beszúrjuk a megfelelő szülő menübe. Az új legördülő menünket a MainMenu1 komponens Items tulajdonságához kell hozzáadnunk. A helyzetét kiszámolhatjuk, hiszen az index nullától indul, vagy lekérhetjük a főmenütől az előző legördülő menü pozícióját:

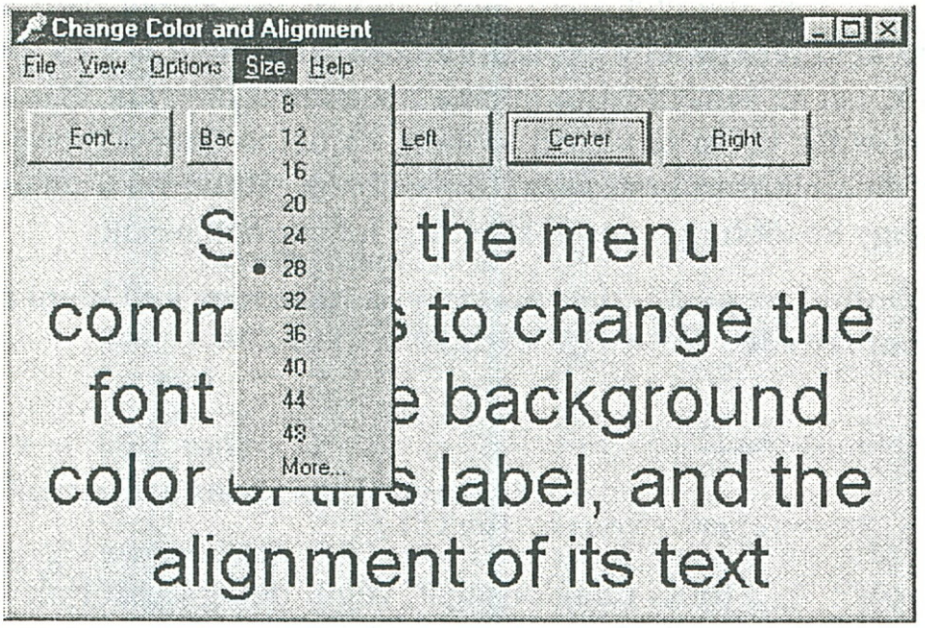
```
Position := MainMenu1.Items.IndexOf (Options1);
MainMenu1.Items.Insert (Position + 1, PullDown);
```

A legördülő menü pontjait egy elöl tesztelő –while- ciklusban készítjük el. Azért nem számlálós – for – ciklust használunk, mert csak minden negyedik méretre akarunk új pontot létrehozni. A létrehozás kódja egy kicsit összetettebb, mert rádiógombokká akarjuk alakítani őket, de az alapfelépítés nagyon egyszerű:

```
var
  Item: TMenuItem;
  I: Integer;
begin
  ...
  I := 8;
```

```
while I <= 48 do
begin
  Item := TMenuItem.Create (self);
  Item.Caption := IntToStr (I);
  PullDown.Insert (PullDown.Count, Item);
  I := I + 4;
end;
```

Új menüpont beszúrásához az Insert metódust használtuk, aminek a menüpontok számát – Pulldown.Count – adtuk paraméterként. A legördülő menü végleges felépítése a 9.8. ábrán látható.



9.8. ábra *A MenuOne3 példa Size legördülő menüje futás alatt keletkezett, a menüpontjaival együtt*

Amint látható, a program még egy extra pontot tesz a menü végére, amivel más, eddig nem listázott fontméreteket állíthatunk be. Az utolsó menüpont OnClick eseményét a Font1Click metódus kezeli le, ami a betűtípus választó dialógust jeleníti meg:

```
Item := TMenuItem.Create (self);
Item.Caption := 'More...';
Item.OnClick := Font1Click;
PullDown.Insert (PullDown.Count, Item);
```

A többi menüpont OnClick eseményét egyetlen metódus kezeli, de ezt az eljárást kézzel kell elkészítenünk, úgy, hogy a deklarációját a form osztályéhoz adjuk:

```
type
  TFormColorText = class(TForm)
    ...
  public
    procedure SizeItemClick(Sender: TObject);
end;
```

A metódusnak a megfelelő paraméterlistával kell rendelkeznie. Íme a Sender paraméteren alapuló metódus kódja:

```
procedure TFormColorText.SizeItemClick(Sender: TObject);
begin
  with Sender as TMenuItem do
    Label1.Font.Size := StrToInt (Caption);
end;
```

Amint az látható, ez a kód nem állítja be a helyes jelzést a kiválasztott elem mellett. Ennek az az oka, hogy a felhasználó új méretet választhat a betűtípus megváltoztatásával is, emiatt más megközelítést fogunk alkalmazni, és a legördülő menü OnClick eseményét kezeljük le. Ez az esemény pont a menü lenyílása előtt aktiválódik, ezért elég ebben beállítani a megfelelő jelölést.

```
procedure TFormColorText.SizeClick (Sender: TObject);
var
  I: Integer;
  Found: Boolean;
begin
  Found := False;
  with Sender as TMenuItem do
  begin
    // look for a match, skipping the last item
    for I := 0 to Count - 2 do
      if StrToInt (Items [I].Caption) =
        Label1.Font.Size then
      begin
        Items [I].Checked := True;
        Found := True;
        System.Break; // skip the rest of the loop
      end;
    if not Found then
      Items [Count - 1].Checked := True;
  end;
end;
```

Ez a részlet végignézi a kiválasztott menü – a Sender, küldő – pontjait, az utolsó kivételével, és megnézi, hogy a címke értéke megegyezik-e az aktuális font méretével. Ha nincsen ilyen pont, akkor a program az utolsó menüpontot állítja be, ezzel jelezve, hogy a más méretet használunk.

Természetesen ezt a SizeClick eseménykezelőt is futásidőben kell beállítanunk, miután a legördülő menüt elkészítettük. Itt következik a FormCreate metódus teljes forrása, ami az összes, ebben a részben tárgyalt lehetőséget összegzi:

```
procedure TFormColorText.FormCreate(Sender: TObject);
var
  PullDown, Item: TMenuItem;
  Position, I: Integer;
begin
  // create the new pulldown menu
  PullDown := TMenuItem.Create (self);
  PullDown.Caption := '&Size';
  PullDown.OnClick := SizeClick;
  // compute the position and add it
  Position := MainMenu1.Items.IndexOf (Options1);
  MainMenu1.Items.Insert (Position + 1, PullDown);
  // create menu items for various sizes
  I := 8;
  while I <= 48 do
  begin
    // create the new item
    Item := TMenuItem.Create (self);
    Item.Caption := IntToStr (I);
    // make it a radio item
    Item.GroupIndex := 1;
    Item.RadioItem := True;
    // handle click and insert
    Item.OnClick := SizeItemClick;
    PullDown.Insert (PullDown.Count, Item);
    I := I + 4;
  end;
  // add extra item at the end
  Item := TMenuItem.Create (self);
  Item.Caption := 'More...';
  // make it a radio item
  Item.GroupIndex := 1;
  Item.RadioItem := True;
  // handle click by showing the font dialog box
  Item.OnClick := Font1Click;
  PullDown.Insert (PullDown.Count, Item);
end;
```

Menük és menüpontok létrehozása dinamikusan

Amikor dinamikusan hozunk létre menüket, vagy menüpontokat, akkor használhatjuk a kapcsolódó komponenseket, ahogy azt a MenuOne3 példában tettük. Másik lehetőségként felhasználhatjuk a Menus egység néhány globális függvényét.

```
function NewMenu(Owner: TComponent; const AName:
string;
  Items: array of TMenuItem): TMainMenu;
function NewPopupMenu(Owner: TComponent;
  const AName: string; Alignment: TPopupAlignment;
  AutoPopup: Boolean; Items: array of TMenuitem):
  TPopupMenu;
function NewSubMenu(const ACaption: string;
  hCtx: Word; const AName: string;
  Items: array of TMenuItem): TMenuItem;
function NewItem(const ACaption: string;
  AShortCut: TShortCut; AChecked, AEnabled: Boolean;
  AOnClick: TNotifyEvent; hCtx: Word;
  const AName: string): TMenuItem;
function NewLine: TMenuItem;
```

A NewMenu és a NewPopupMenu függvényeket általánosan új menük létrehozására használhatjuk. A megfelelő osztályok konstruktorának meghívása a valóságban nem mindig működik helyesen.

Rövid és hosszú menük

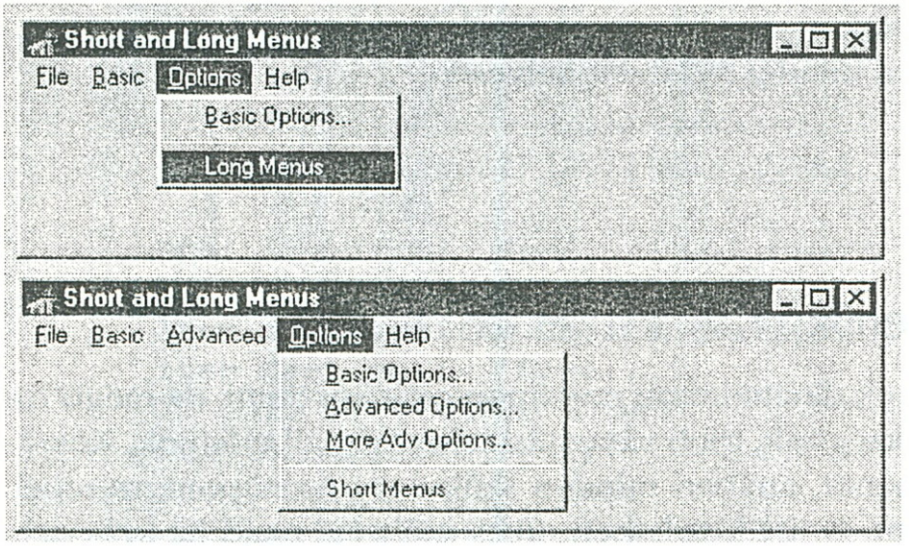
Amennyiben nem akarunk dinamikusan menüpontokat készíteni, de mégis rugalmas menüre van szükségünk, több választási lehetőség áll előttünk. Készíthetünk egy hatalmas menüt, amiben minden szükséges menüpont szerepel, azután elrejtünk mindent, amire kezdetben nincs szükség, majd ha kellenek, akkor megmutatjuk őket. Ez a megoldás az előbb ismertetetthez áll közel.

Készíthetünk még több menüt, néhol azonos elemekkel, és ezeket szükség szerint cserélgethetjük. Ezzel a megközelítéssel foglalkozik ez a rész. Erre tipikus példa, ha két különböző felhasználói réteg – a kezdők és a tapasztaltak számára két különböző menüt készítünk, egy rövidet, és egy hosszút. Ez a technika néhány éve még sok nagy Windows alkalmazásban megfigyelhető volt, de mára már más megközelítések, mint például a felhasználó által teljesen átszerkeszthető menük, teljesen kiszorították.

Maga az ötlet egyszerű, és a megvalósítása is magától értetődő:

1. Készítsük el az alkalmazás teljes menüjét, és adjunk hozzá egy menüpontot „Short" (rövid) néven
2. Adjuk ezt a menüt a Delphi menü sablonok közé.
3. Tegyünk egy másik MainMenu komponenst is a formra, aminek a struktúráját a sablonból másoljuk.
4. A második menüből távolítsunk el minden összetettebb funkcióra mutató menüpontot, és a különleges elem nevét „short"-ról módosítsuk „long"-ra (hosszú).
5. A form Menu tulajdonságánál állítsuk be a két lehetséges közül azt a MainMenu komponenst, amit az alkalmazás indításakor látni szeretnénk. Ennek a beállításnak a hatása meglátszik a tervezés alatt is.
6. Írjuk meg a Short és a Long parancsokat, amiknek hatására a menük kicserélődnek.

Ha ezeket a lépéseket követjük, egy, a 9.9. ábrán látható TwoMenus alkalmazáshoz nagyon hasonló programunk lesz, ami menet közben tudja változtatni a menüjét. A példának két MainMenu komponense van, használhatatlan „buta" menüpontokkal, és a Short és Long parancsokkal.



9.9. ábra *A TwoMenus példa rövid és hosszú menüje*

Az alkalmazás a Short Menu és Long Menu pontok kiválasztásánál bekövetkező menücserén kívül semmit nem csinál. Itt a Short Menu kódja:

```
procedure TForm1.ShortMenus1Click(Sender: TObject);
begin
  Form1.Menu := MainMenu2;    //activate short menu
end;
```

Grafikus menüpontok

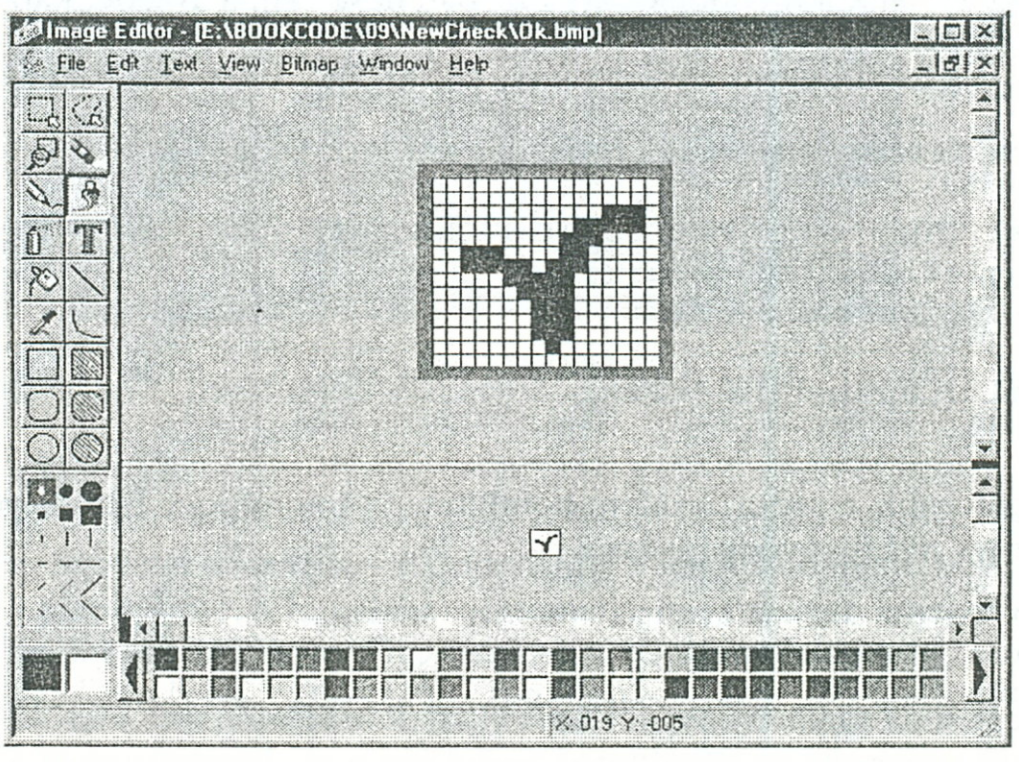
Amellett, amiről eddig szó volt, miszerint hogy futásidőben megváltoztathatjuk a menü struktúráját (amit közvetlenül Delphiből meg is tehetünk), van még számos olyan művelet, amiket a Windows API segítségével végezhetünk a menükön, hiszen azokkal számtalan API függvény foglalkozik.

Windows API függvényekkel grafikát tehetünk a menükbe, megváltoztathatjuk a pipa alakját, kicserélhetjük a szövegeket képekre, és még rajzolhatunk is a menüpontokba.

A pipa módosítása

Ahogy azt az előbb említettük, Windowsban többféleképpen módosíthatjuk a menüket. Ebben a részben azt nézzük meg, hogyan tudjuk átalakítani menüpontok melletti pipát két saját kép segítségével. Ez a példa, a NewCheck, képeket használ, és egy Windows API függvényt hív.

Először azt nézzük meg, hogy miért kell két kép, és nem csak egy, hiszen, ha egy menüpontot megnézünk, akkor ott vagy egy pipa van, vagy nincs semmi. A valóságban azonban a Windows két különböző bittérképet használ a jelölt és a jelöletlen menüpontokra. A példában szereplő két 16 × 14 pixeles bittérképet a Delphi Image Editor eszközével készítettük, ami a 9.10. ábrán is látható. Ezt a **Tools** menüből érhetjük el, de bármilyen másik szerkesztővel is előállíthatjuk a képeinket, beleértve a Windows Paintbrusht is. A bittérképeket két BMP állomány formájában a projekt mappájában tároljuk.



9.10. ábra *A két új jelölés egyike a Delphi Image Editorban*

A NewCheck példa formja nagyon egyszerű, csak két komponenst tartalmaz, egy főmenüt, és egy cimkét:

```
object Form1: TForm1
  Caption = 'New Check'
  Menu = MainMenu1
  OnCreate = FormCreate
  OnDestroy = FormDestroy
  object Label1: TLabel
    Alignment = taCenter
    AutoSize = False
    Caption = 'OFF'
    Font.Height = -96
    Font.Name = 'Arial'
    Font.Style = [fsBold]
  end
  object MainMenu1: TMainMenu
    object Command1: TMenuItem
      Caption = '&Command'
      OnClick = Command1Click
      object Toggle1: TMenuItem
        Caption = '&Toggle'
        OnClick = Toggle1Click
      end
    end
  end
end
```

Amint az a listából látható, a menünek csak egy parancsa van, a Toggle, kapcsolás, amivel a címke szövegét változtathatjuk meg „ON"-ról – bekapcsolt – „OFF"-ra – kikapcsolt –, és ezzel együtt a jelölés is változik.

```
procedure TForm1.Toggle1Click(Sender: TObject);
begin
  Toggle1.Checked := not Toggle1.Checked;
  if Toggle1.Checked then
    Label1.Caption := 'ON'
  else
    Label1.Caption := 'OFF';
end;
```

A program legfontosabb része a SetMenuItemBitmaps (menüpont bittérkép beállítása) Windows API függvény hívása.

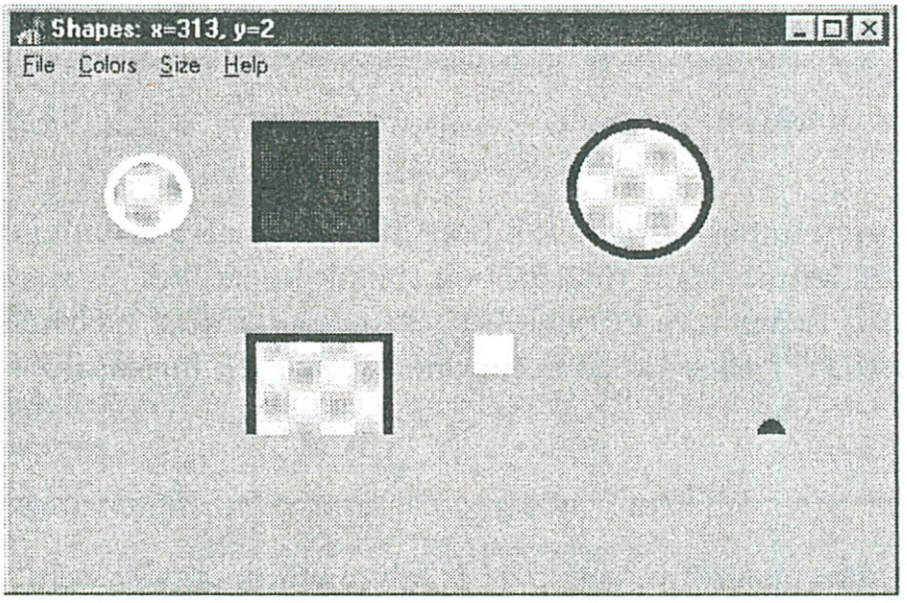
```
function SetMenuItemBitmaps (Menu: HMenu;
  Position, Flags: Word;
  BitmapUnchecked, BitmapChecked: HBitmap): Bool;
```

Ennek a függvénynek a következő paraméterei vannak:

- Az első paraméter az a legördülő menü, amire hivatkozunk.
- A második a menüpont legördülő menün belüli sorszáma.
- A harmadik paraméter egy jelző, ami eldönti, hogy hogyan kell értelmezni a második paramétert.
- Az utolsó két paraméter pedig a két használandó bittérkép.

Ebből látszik, hogy a jelző beállítása csak egy meghatározott menüpontra vonatkozik. A következő kód bemutatja, hogy hívhatjuk meg Delphiből a függvényt. Példaként lásd a 9.11. ábrát.

```
procedure TForm1.Command1Click(Sender: TObject);
begin
  SetMenuItemBitmaps (Command1.Handle,
    Toggle1.Command, MF_BYCOMMAND,
    Bmp2.Handle, Bmp1.Handle);
end;
```



9.11. ábra *A NewCheck példa futás közben. Vegyük észre az új pipát*

Ez a hívás a kódban korábban definiált bittérkép változót, és néhány komponens nevét használja – a Command1 a legördülő menü neve, a Toggle1 pedig a menüponté. A fenti kód megmutatja, hogy milyen egyszerű egy elem leíróját átadni egy Windows függvénynek a Handle tulajdonság használatával.

Először arra gondoltunk, hogy ezt a függvényt akkor kellene meghívni, amikor a form elkészül, és a két képet betöltöttük az állományokból, de ez nem jó megoldás. A Delphi némileg megváltoztatja a Windows viselkedését,

ráerőltetve az alkalmazásra, hogy a menü megjelenítésekor mindig újra hozzárendelje a bittérképeket a menüpontokhoz. Erre azt a megoldást találtuk, hogy a függvényt mindig meghívjuk, akárhányszor a legördülő menüt kiválasztjuk, ezért használjuk a legördülő menü OnClick eseményét.

Már csak az van hátra, hogy betöltsük a képeket. Ehhez hozzá kell adnunk két TBitmap (Tbittérkép) típusú mezőt a form osztályához, létre kell hoznunk a két objektum egy-egy egyedét, és ezután be kell töltenünk a képeket a két BMP állományból. Ezt csak egyszer, a form készítésekor kell megtennünk:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Bmp1 := TBitmap.Create;
  Bmp2 := TBitmap.Create;
  Bmp1.LoadFromFile ('ok.bmp');
  Bmp2.LoadFromFile ('no.bmp');
end;
```

A program kilépésekor, vagyis a form OnDestroy eseménykezelőjében, a két képet meg kell semmisíteni:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Bmp1.Free;
  Bmp2.Free;
end;
```

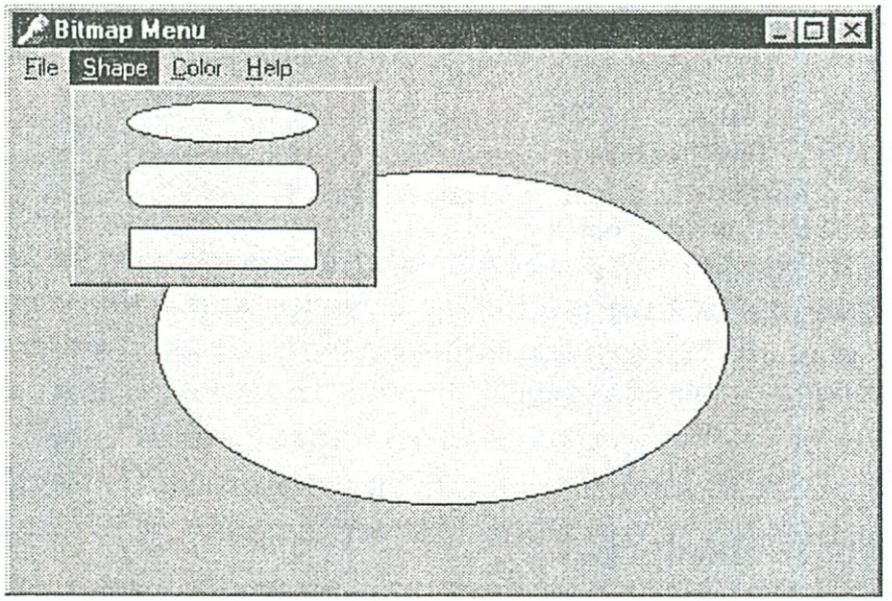
Ahhoz, hogy ezt a programot futtathassuk, a két BMP állománynak a futtatható állomány mappájában kell lennie, mert futásidőben töltjük be azokat, és nem a Delphi EXE állományába beágyazva tároljuk. Másik megoldásként használhattunk volna két nem látható Image (kép) komponenst a képek tárolására.

Megjegyzés

Képeket az alkalmazás erőforrásai közé is felvehetünk, ami bekerülhet a futtatható állományba, és így az alkalmazást egyetlen állományként terjeszthetjük. Ez az eljárás azonban összetett, ezért most nem mutatjuk be. Későbbi fejezetek, a tizenegyediktől kezdődően, tartalmaznak példákat erőforrások között tárolt képekre, a huszonhetedik fejezetet pedig teljes egészében a Windows erőforrásoknak szenteltük.

Kép menüpontok

Menüpontok Checked állapotát jelző bittérképek használata mellett, akár a menüpont szövegét is kicserélhetjük egy bittérképpel. Bizonyos esetekben alkalmazásunk ettől könnyebben használhatóvá válik. A 9.12. ábra az elkészítésre kerülő BitMenu példa egy bittérképes menüjét ábrázolja.



9.12. ábra *A BitMenu példa Shape legördülő menüje képeket használ karakterláncok helyett*

A BitMenu program nagyon egyszerű. A form közepén egy alakzat látható, aminek a színét és formáját a menüben állíthatjuk be – ez lehet téglalap, lekerekített téglalap, vagy ellipszis. Itt következik a form és a menü szöveges leírása:

```
object Form1: TForm1
  Caption = ' Bitmap Menu'
  Menu = MainMenu1
  OnCreate = FormCreate
  OnDestroy = FormDestroy
  OnResize = FormResize
  object ShapeDemo: TShape...   // default properties
  object MainMenu1: TMainMenu
    object File1: TMenuItem...
      object Exit1: TMenuItem...
          end
    object Shape1: TMenuItem
      Caption = '&Shape'
      object Ellipse1: TMenuItem
        Caption = 'Ellipse'
```

```
        OnClick = Ellipse1Click
      end
      object RoundRec1: TMenuItem
        Caption = 'RoundRec'
        OnClick = RoundRec1Click
      end
      object Rectang1: TMenuItem
        Caption = 'Rectang'
        OnClick = Rectang1Click
      end
    end
    object Color1: TMenuItem
      Caption = '&Color'
      object Red1: TMenuItem
        Caption = 'Red'
        OnClick = Red1Click
      end
      object Green1: TMenuItem
        Caption = 'Green'
        OnClick = Green1Click
      end
      object Blue1: TMenuItem
        Caption = 'Blue'
        OnClick = Blue1Click
      end
    end
    object Help1: TMenuItem...
      object About1: TMenuItem...
  end
end
```

Azért szerepel a menüpontok teljes leírása, mert, ahogy az hamarosan látni fogjuk, a megnevezésük fontos szerepet játszik a kódban. Második lépésként a programot működővé tettük a különböző menüparancsok lekezelésével, amire most két példát mutatunk:

```
procedure TForm1.Red1Click(Sender: TObject);
begin
  ShapeDemo.Brush.Color := clRed;
end;
procedure TForm1.Ellipse1Click(Sender: TObject);
begin
  ShapeDemo.Shape := stEllipse;
end;
```

A form OnResize eseményéhez is készítettünk egy kezelőt, ami a form méretéhez igazítja az alakzat méretét. A négy pozíciót jelző tulajdonság – Left, Top, Width, és Height – állítása helyett a SetBounds (határok beállítása) metódust hívtuk meg, ami gyorsabb programot eredményez.

Miután elkészítettük a program kódját, itt az idő, hogy a menüpontokat bittérképekké alakítsuk. Ehhez mind a három mintához és színhez külön kép állományra lesz szükségünk. Bár ezek a képek tetszőleges méretűek lehetnek, érdemes szélesre és laposra rajzolni őket, hogy hasonlítsanak egy menüpontra.

Miután elkészült a hat darab képállomány, mindegyik a saját menüpontjának megfelelő névvel, és BMP kiterjesztéssel, készítsük el a form OnCreate eseményét, amiben az ál-menüpontok szövegét kicserélhetjük a képekre. Erre a ModifyMenu (menü módosítása) API függvény használható:

```
function ModifyMenu (hMnu: HMENU;
  uPosition, uFlags, uIDNewItem: UINT;
  lpNewItem: PChar): BOOL; stdcall;
```

Az első paraméter a módosítandó menü, általában legördülő menü leírója, a második a módosítandó menüpont pozíciója, a harmadik jelzőket tartalmaz, amik a többi paraméter hatását módosítják, a negyedik a menüpont azonosítója, amit a Delphi a Command (parancs) tulajdonságban tárol, és az utolsó paraméter az új menüpont szövege. Hogy hogyan használhatjuk fel ezt kép beállítására? Egyszerűen, egy kép leíróját adjuk át utolsó paraméterként szöveg helyett, és az uFlags paraméter értékében az mf_Bitmap jelzőt használjuk.

Most akkor megnézhetjük a forráskódot, ami két ciklust tartalmaz. Mivel ez a lehetőségekhez képest általános kód, ezért új szín vagy forma hozzáadása nagyon egyszerű.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
  Bmp: TBitmap;
begin
  // load the bitmaps for the shapes
  for I := 0 to Shape1.Count - 1 do
  begin
    Bmp := TBitmap.Create;
    Bmp.LoadFromFile (Shape1.Items [I].Caption+'.bmp');
    ModifyMenu(Shape1.Handle,Shape1.Items[I].MenuIndex,
      mf_ByPosition or mf_Bitmap,
      Shape1.Items [I].Command, Pointer (Bmp.Handle));
    Shape1.Items [I].Tag := Integer (Bmp);
  end;

  // load the bitmaps for the colors
  for I := 0 to Color1.Count - 1 do
  begin
    Bmp := TBitmap.Create;
    Bmp.LoadFromFile (Color1.Items [I].Caption+'.bmp');
    ModifyMenu(Color1.Handle,Color1.Items[I].MenuIndex,
      mf_ByPosition or mf_Bitmap,
```

```
            Color1.Items[I].Command,
            Pointer (Bmp.Handle));
          Color1.Items [I].Tag := Integer (Bmp);
      end;
    end;
```

Tipp

A fenti kódban a ModifyMenu API függvény hívásakor a mf_ByPosition (pozíció szerint) jelzőt használtuk a harmadik paraméterben, és második paraméterként átadtuk a menüpont helyét. Emellett azonban még át kell adnunk a menüparancsot is a negyedik paraméterben, mert ez lesz az új menüpont parancsa. Ha nem így tennénk, elveszítenénk a kapcsolatot a Delphi eseménykezelő és a menüpont között.

Láthattuk, hogy a bittérkép objektumok hivatkozásai elmentésre kerültek a hozzájuk kapcsolódó menüpont Tag tulajdonságában. Ez azért történt így, hogy a program befejezésekor ezeket az objektumokat megsemmisíthessük.

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to Shape1.Count - 1 do
    TBitmap (Shape1.Items [I].Tag).Free;
  for I := 0 to Color1.Count - 1 do
    TBitmap (Color1.Items [I].Tag).Free;
end;
```

Másik megoldásként a bittérkép objektumokat tárolhattunk volna egy, a form mezőjeként deklarált tömbben, és a végén megsemmisíthettük volna a tömb összes elemét.

Amikor ezt a programot futtatjuk, a Shape (formák) legördülő menü jól néz ki, a Color (Színek) menü azonban nem. Amikor kiválasztjuk a menüt, a színek még jók, de ha ráállunk egy menüpontra, azt a színt a menü kiválasztás a negatívjára változtatja. A kiválasztott szín negatívra változtatása pedig elég furcsa eredményeket hoz, hiszen, például ahhoz, hogy a kék színt kiválaszthassuk egy olyan pontra kell kattintanunk, ami éppen akkor változott sárgává!

Általában célszerű csak fekete-fehér, vagy szürke menüpontokat használni, mert a színes képek sok problémát jelentenek. Ha mégis ilyet akarunk, akkor arra tulajdonos által rajzolt menüpontokat használjunk, amikről a következő részben lesz szó.

Tulajdonos által rajzolt menüpontok

Windowsban a listadobozok, gombok, szöveg mezők, menüpontok, és hasonló alkotóelemek kirajzolása általában a rendszer feladata, hiszen ezek a kontrolok tudják, hogy hogyan fessék újra magukat. Másik lehetőségként, a rendszer lehetővé teszi a kontrolok tulajdonosa, általában a form, számára, hogy az rajzolja ki őket. Ezt, a gombok, listadobozok, kombi panelek, és menüpontok esetében használható technikát hívjuk tulajdonos által rajzoltnak, vagy owner-draw-nak.

A valóságban Delphiben a helyzet némiképp összetettebb. A komponensek megrajzolhatják magukat itt is, ahogy az a TBitBtn képes gombok esetében történik, és még ide kapcsolódó események is vannak. Ha nem gondolunk bele a belső részletekbe, akkor a lista dobozok, kombi panelek tulajdonos általi megrajzolása csak egy csomó eseménykezelő megírását jelenti.

A Delphi azonban nem támogatja a tulajdonos által rajzolt menüpontokat. Ebben az esetben a hagyományos Windows megközelítést kell alkalmaznunk, és a formunkon le kell kezelnünk több rendszer üzenetet, aminek a menetét az 5. fejezetben ismerhettük meg. Itt következik ezeknek a metódusoknak a definíciója az ODMenu példából, ami az előző részben megbeszélt BitMenu példa továbbfejlesztése:

```
type
  TForm1 = class(TForm)
    ...
  public
    procedure WmMeasureItem (var Msg: TWmMeasureItem);
      message wm_MeasureItem;
    procedure WmDrawItem (var Msg: TWmDrawItem);
      message wm_DrawItem;
  end;
```

A wm_MeasureItem üzenetet a Windows legördülő menü kinyitásakor minden egyes menüpontra elküldi egyszer, hogy megállapítsa az elemek méretét. A wm_DrawItem üzenet akkor keletkezik, ha egy menüpont kirajzolásra kerül. Ez olyankor történik, amikor a Windows először megjeleníti a menüpontokat, és valahányszor a menüpont státusza megváltozik; például amikor az egér felette van, akkor ki kell emelni. A menüpontok kirajzolásánál figyelnünk kell minden lehetőségre, beleértve a kiemelt elem más színekkel való megrajzolását, a szükséges pipák megjelenítését, és így tovább.

Az ODMenu példában lekezeljük ugyan a kiemelést, de a többi szolgáltatást, mint a pipák, elhagyjuk. Módosítsuk a CreateForm metódust, hogy az meghívja a ModifyMenu API függvényt, ami átadja az mfOwnerDraw menüjelzőt. Utolsó paraméterként egy kódot adunk át, hogy meg lehessen különböztetni a különböző elemeket. A többi paraméter megegyezik az előző példában használtakkal:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  ...
  // turn the menu items into owner-draw items
  for I := 0 to Color1.Count - 1 do
  begin
    ModifyMenu(Color1.Handle,Color1.Items[I].MenuIndex,
      mf_ByPosition or mf_OwnerDraw,
      Color1.Items[I].Command,Pointer (I));
  end;
end;
```

Most írjuk meg a két, tulajdonos általi rajzoláshoz szükséges üzenetkezelőt. A WmMeasureItem metódus TwmMeausreItem típusú paraméterként egy mutatót kap egy Windows MeasureItemStruct struktúrára, aminek a szerkezetét a Windows API súgóban találhatjuk meg. Ebből a struktúrából a CtlType mezőt fogjuk használni, ami a lemérendő elem típusát tárolja, és amiből megállapíthatjuk, hogy ez a művelet kapcsolatban áll-e menüvel. Ha igen, akkor az ItemWidth (tárgy szélesség) és ItemHeight (tárgymagasság) mezőkben beállítjuk az elem szélességét és magasságát. Mivel az összes elem ugyanolyan méretű, ezért ez a kód nagyon egyszerű. Az utolsó teendőnk, hogy az üzenetnek visszatérési értéket adjunk, jelezve, hogy lekezeltük. Ezt az üzenet struktúra Result (eredmény) mezőjében tehetjük meg. Íme a teljes kód:

```
procedure TForm1.WmMeasureItem (var Msg:
TWmMeasureItem);
begin
  inherited;
  with Msg.MeasureItemStruct^ do
    if CtlType = odt_Menu then
    begin
      ItemWidth := 80;
      ItemHeight := 30;
      Msg.Result := 1; // we've handled it
    end;
end;
```

A menüpontok kirajzolása némileg bonyolultabb, mivel ahhoz, hogy Delphi kódot írhassunk, készítenünk kell egy TCanvas objektumot, ami magába foglalja a Windows eszköz környezet leíróját. Mivel a végén ezt az objektumot a hatékony memóriakezelés érdekében fel kell szabadítanunk, még akkor is, ha hiba történik, ezért egy `try-finally` kifejezést fogunk alkalmazni:

```
Canvas1 := TCanvas.Create;
Canvas1.Handle := hDC;
try
```

```
    ...
  finally
    Canvas1.Free;
  end;
```

Ez a kódrészlet a TCanvas objektum Handle tulajdonságának értékül adja az üzenet által átadott DrawItemStruct struktúra hDC mezőjét. Az üzenetkezelő által megkapott TWmDrawItem paraméter egyik mezőjében van erre a struktúrára egy mutató. Ez a hDC mező adja át a kódunknak a kirajzolni kívánt menüpont eszköz környezetének – a rajzolási területnek, a vászonnak – a leíróját.

Miután beállítottuk a rajzolási mechanizmust, elkezdhetjük a tényleges kódot. Először meg kell állapítanunk a menüpont állapotát, amit az ItemState (tárgy állapota) mezőben találunk. Ha ebben szerepel az ods_Selected (Kiválaszott) jelző – aminek a jelenlétét bitenkénti „és" művelettel ellenőrizhetjük –, akkor a menüpont hátterét a Windows kiemelt összetevők számára kijelölt rendszerszínével, a clHighlight-tal kell kirajzolnunk, máskülönben a normál menü színt használjuk, a clMenu-t.

Megjegyzés

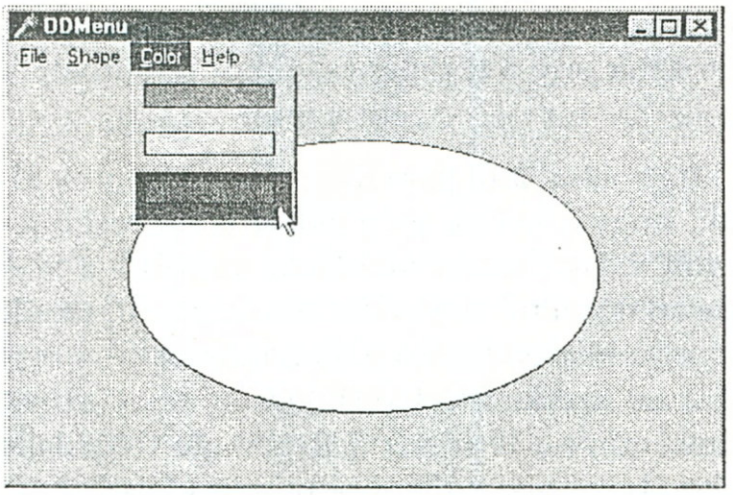
Ahogy arról a hetedik fejezetben szó volt, néhány Delphi színkonstans megfelel a Windows rendszer beállításainak. Ezek nem állandó színek, hanem a felhasználó beállításait tükrözik.

Miután a vászon (Canvas) ecsetéhez (Brush) hozzárendeltük a háttérszínt, könnyedén kifesthetjük a FillRect metódus segítségével. Ennek a metódusnak egyetlen paramétere van, ami a törölni kívánt téglalapra vonatkozik. Ez az információ a DrawItemStruct struktúra rcItem mezőjében található. Figyeljünk erre, mert ha a rajzolást nem korlátozzuk erre a téglalapra, akkor felülírhatjuk a többi menüpontot is!

A következő lépés a színes részek kirajzolása, ami nem függ a menüpontok állapotától, hiszen éppen azt akartuk, hogy a színek még akkor is helyesen jelenjenek meg, ha a menüpont ki van választva. Az ItemData mezőből a kód visszakapja a menüpontok felhasználó által rajzolttá tételéhez használt ModifyMenu függvényhívás utolsó paraméterét. Ennek a mezőnek az értékétől függően a program egy egyszerű case szerkezettel beállítja az ecset színét (Color). Ezután már csak ki kell festenünk a területet a Rectangle (téglalap) függvénnyel, aminek négy paraméterében, egy, a menüpont teljes felszínénél kisebb területet adunk meg, hiszen szükségünk van háttérszínű keretre, ami a példánkban öt képpont széles.

Végül itt áll a WmDrawItem metódus teljes forrása:

```
procedure TForm1.WmDrawItem (var Msg: TWmDrawItem);
var
  Canvas1: TCanvas;
begin
  inherited;
  with Msg.DrawItemStruct^ do
    if CtlType = odt_Menu then
    begin
      // create a canvas for painting
      Canvas1 := TCanvas.Create;
      Canvas1.Handle := hDC;
      try
        // set the background color and draw it
        if (ods_Selected and ItemState <> 0) then
          Canvas1.Brush.Color := clHighlight
        else
          Canvas1.Brush.Color := clMenu;
        Canvas1.FillRect (rcItem);
        case ItemData of
          0: Canvas1.Brush.Color := clRed;
          1: Canvas1.Brush.Color := clLime;
          2: Canvas1.Brush.Color := clBlue;
        end;
        Canvas1.Rectangle (rcItem.Left + 5,
          rcItem.Top + 5,
          rcItem.Right - 10, rcItem.Bottom - 10);
      finally
        Canvas1.Free;
      end;
    end;
end;
```



9.13. ábra *Tulajdonos által rajzolt legördülő menük az ODMenu példában*

Ennek a kódnak a hatását a 9.13. ábrán láthatjuk. A színes menüpontok kirajzolásában megjelenő különbségek megnézéséhez futtassuk egyszerre ezt a programot a BitMenu példával. A két példát nézve megpróbálkozhatunk azzal, hogy a fekete fehér képes menüpontokat is a felhasználó által rajzolttá tegyük.

A rendszermenü testreszabása

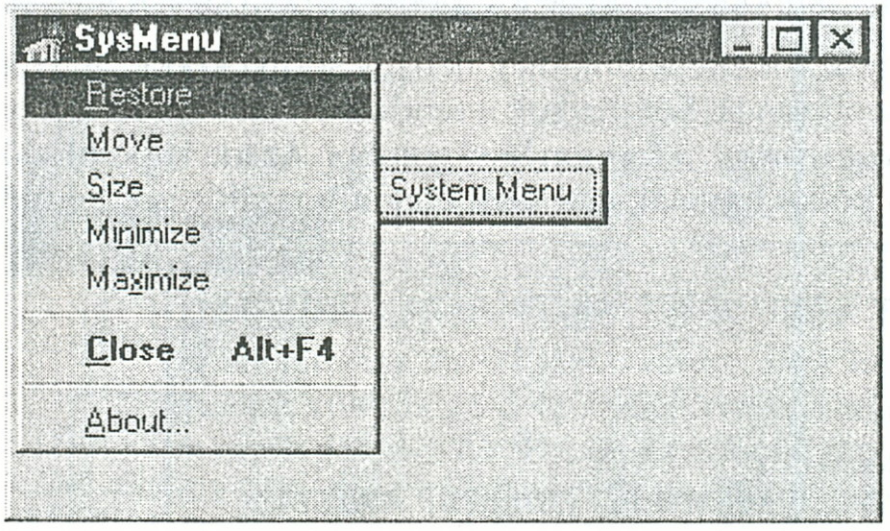
Néha érdekes lehet, hogy menüpontokat adjunk a rendszer menühöz, a menüsor helyett, vagy amellett. Ez hasznos lehet másodlagos ablakoknál, eszköztáraknál, nagyméretű ablakoknál, és „gyorsat-gyorsan" típusú alkalmazásoknál. Egy menüpont hozzáadása a rendszer menühöz egyszerű:

```
AppendMenu (GetSystemMenu (Handle, FALSE),
  MF_SEPARATOR, 0, '');
AppendMenu (GetSystemMenu (Handle, FALSE),
  MF_STRING, idSysAbout, '&About...');
```

A SysMenu példából származó fenti kódrészlet egy elválasztót, és egy új menüpontot ad a rendszermenühöz. A GetSystemMenu API hívásnak a form leírójára van szüksége, és a rendszer menü leíróját adja vissza. Az AppendMenu API függvény általános célú függvény, amivel menüpontokat, vagy akár egész legördülő menüket adhatunk bármely menühöz, legyen az a menüsor, a rendszer menü, vagy egy már meglévő legördülő menü. Új menüpont hozzáadásakor meg kell adnunk a szövegét, és az azonosító számát. A példában ezt az azonosítót a következő módon definiáltuk:

```
const
  idSysAbout = 100;
```

A SysMenu példában ez a kód szerepel az OnCreate eseménykezelőben, és a 9.14. ábrán látható rendszer menüt eredményezi.



9.14. ábra *A Sysmenu példa rendszermenüjének új pontját a form OnCreate eseménykezelője készítette*

Menüpontot hozzáadni könnyű, de hogyan kezeljük le? Normál menüpontok kiválasztásakor wm_Command Windows üzenet keletkezik, amit a Delphi lekezel, és meghívja a megfelelő menüpont OnClick eseménykezelőjét. A rendszermenü pontjainak kiválasztásakor azonban wm_SysCommand üzenet keletkezik, ami a Delphi általános kezelőjéhez kerül. A rendszer menü parancsaira általában a Windowsnak kell válaszolnia.

Elfoghatjuk ezt az üzenetet, és megnézhetjük, hogy a parancs azonosítója, ami a TWmSysCommand paraméter CmdType mezőjében érkezik, megegyezik-e a mi idSysAbout azonosítónkkal. Mivel a Delphiben nincs ennek megfelelő metódus, ezért nekünk kell definiálnunk egy új üzenetre válaszoló metódust a form osztályába:

```
public
  procedure WMSysCommand (var Msg: TMessage);
    message wm_SysCommand;
```

Az eljárás kódja nem túl összetett, csak megnézzük, hogy a parancs nekünk szól-e, és meghívjuk az általános kezelőt:

```
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);
begin
  if Msg.CmdType = idSysAbout then
    ShowMessage ('Mastering Delphi: SysMenu example');
  inherited;
end;
```

Bonyolultabb rendszermenü készítésekor ahelyett, hogy a fenti módszerrel minden menüpontot lekezelnénk, másik megközelítést is alkalmazhatunk. Tegyünk a formra egy MainMenu komponenst, és készítsük el a felépítését – hogy milyen, az lényegtelen-, és írjuk meg a szükséges eseménykezelőket. Ezután állítsuk vissza a form Menu tulajdonságát, ezzel eltávolítva a menüsort. Így, ugyan van MainMenu komponensünk, de a képernyőn nem látszik semmi.

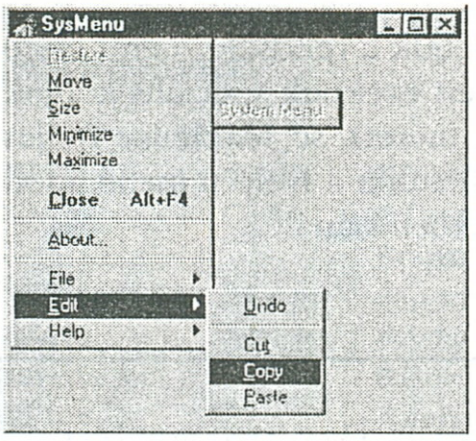
Ezután a SysMenü példához hozzátehetünk még néhány sornyi kódot, amivel a rejtett menü minden menüpontját hozzáadjuk a rendszermenühöz. Ez akkor következik be, amikor megnyomjuk a formon lévő gombot. Az ide kapcsolódó kezelő teljesen általános kódot használ, ami nem függ a hozzáfűzésre kerülő menü felépítésétől:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  // add a separator
  AppendMenu (GetSystemMenu (Handle, FALSE),
    MF_SEPARATOR, 0, '');
  // add the main menu to the system menu
  with MainMenu1 do
```

```
    for I := 0 to Items.Count - 1 do
      AppendMenu (GetSystemMenu (self.Handle, FALSE),
        mf_Popup, Items[I].Handle,
        PChar (Items[I].Caption));
  // disable the button
  Button1.Enabled := False;
end;
```

Ez a kódrészlet a self.Handle kifejezést használja, hogy elérje a form leíróját, erre azért volt szükség, mert a „with" kifejezés miatt éppen a MainMenu1 komponensen dolgoztunk.

Az itt használt menüjelző, a mf_Popup azt jelzi, hogy legördülő menüt adunk a rendszer menühöz. A függvény negyedik paramétere a hozzáadandó legördülő menü leírója – az előző példában e helyett a menü azonosítóját adtuk hozzá. Mivel a rendszer menühöz almenükkel rendelkező pontokat adunk, ezért a rendszer menü végleges struktúrája kétszintes lesz, ahogy az a 9.15. ábrán látható.



9.15. ábra *A SysMenu példa második szinten lévő menürendszere úgy keletkezett, hogy a teljes főmenüt bemásoltuk a rendszer menübe*

Figyelem!

A Windows API az előugró és a legördülő menü kifejezéseket felváltva alkalmazza. Ez fölöttébb furcsa, mert legtöbbünknek ezek a kifejezések különböző dolgot jelentenek; a helyi menüket, és a menüsor másodlagos menüit. Ebből, és abból a tényből, hogy belső felépítésük is azonos, arra lehet következtetni, hogy valószínűleg ugyanarra a komponensre épültek.

Miután hozzáadtuk a menüpontokat a rendszermenühöz, le is kell kezelnünk azokat. Természetesen ellenőrizhetjük az összes menüpontot a WMSysCommand metódusban, vagy kereshetünk jobb megoldást. Mivel Delphiben egyszerűbb minden ponthoz OnClick eseménykezelőt írni, ezért inkább megkeressük a menü struktúrában a kapott azonosítóhoz tartozó parancsot. A Delphi itt a FindItem metódussal segít.

Ha megtaláltuk a menüpontot, akkor meghívjuk a Click metódusát, ami meghívja az OnClick kezelőt. Íme a WMSysCommand metódushoz írt kód:

```
var
  Item: TMenuItem;
begin
  ...
  Item := MainMenu1.FindItem (Msg.CmdType, fkCommand);
  if Item <> nil then
    Item.Click;
```

Ebben a kódban a paraméterként megkapott üzenetstruktúra CmdType mezője tartalmazza a meghívott menüpont parancsát.

Egyszerű „if" vagy „case" szerkezetekkel lekezelhetjük a rendszer menü előre definiált menüpontjait is ezzel az azonosítóval, mert ezeknek speciális kódjuk van, mint a sc_Close (Bezárás), sc_Minimize (kis méret), sc_Maximize (teljes méret), és így tovább. Bővebb információkat találhatunk a Delphi mellé adott Windows API súgóban a wm_Syscommand üzenet leírásánál.

Megjegyzés

Ez az alkalmazás működik, de van egy hibája. Ha a jobb gombbal az alkalmazást jelképező ikonra kattintunk a Tálcán, akkor egy egyszerű rendszer menüt kapunk, ami még az alapértelmezettől is eltér. Ez azért van, mert ez a rendszermenü egy másik ablakhoz tartozik, a globális Application (alkalmazás) objektum ablakához. Az Application objektumról a huszonötödik fejezetben lesz szó, ahol elkészítjük ennek a példának a Tálca ikonnal is működő változatát.

Teljes menü készítése

Most, hogy tudjuk, hogyan készíthetünk menüt, hogyan kapcsolhatjuk ki, és jelölhetjük meg a menüpontokat, elkészíthetjük egy teljes alkalmazás menüjét. Ez előző fejezetben ismertetett RichNote példa egy RichEdit komponensre épülő egyszerű szövegszerkesztő volt, amivel írhattunk, és megváltoztathattunk a

szöveg betűtípusát, de csak ennyit tudott. Most menüt adnunk hozzá, és kifejlesztünk számos szolgáltatást, többek között a szövegállományok megnyitására és elmentésére vonatkozókat. Ez annyit jelent, hogy a végére képessé tesszük a programunkat arra, hogy új állomány megnyitása előtt megkérdezze a felhasználót, hogy elmentse-e a változásokat.

Először el kell készítenünk, egy, a konvenciókat követő menüt. A főmenü két hagyományos menüpontokat tartalmazó szokásos menüvel kezdődik, ez a Fájl (File), és a Szerkesztés (Edit). Ezt követi két, az alkalmazásra jellemző menü, a Betűtípus (Font) és a Bekezdés (Paragraph), amik a betűtípus és a szöveg igazítás beállítására szolgálnak.

Az utolsó két menü, a Beállítások (Options), és a Súgó (Help) majdnem szabványosak; a nevük a megszokott, de a tartalmuk nem. A Beállítások menü a háttérszín állítására, és a karakterek megszámolására használható, a Súgóban pedig csak egy Névjegy (About) van.

Ebben a példában a menüsor legtöbb parancsát elkészítjük, de nem mindet. A Vágólapot használó parancsokat – vagyis a Szerkesztés menü parancsait – a harmincadik fejezetben készítjük el. A következő táblázat mutatja a menü teljes szerkezetét:

&File	**&Edit**	**F&ont**	**&Paragraph**	**&Options**	**&Help**
&New	Cu&t	&Times New Roman	&Left Aligned	&Background Color	&About RichNote...
&Open...	&Copy	&Courier New	&Right Aligned	&Read Only	
&Save	&Paste	&Arial	&Centered	&Count chars...	
Save&As...		&Bold			
&Print...		&Italic			
E&xit		&Small			
		&Medium			
		&Large			
		More &Fonts...			

A teljes menü elkészítése után megszabadulhatunk a RichNote példa egyszerű panelétől, és a betűtípus gombtól. A form egyetlen vizuális komponense, amit a RichNot2 változatban meghagyunk a RichEdit komponens, amit a kliens területhez igazítunk. Azután lesz még a Főmenü komponens, és négy szabványos dialógusablak – az OpenDialog (Megnyitás), a SaveDialog (Mentés), a FontDialog (Betűtípus), és a ColorDialog (Színek).

A Fájl menü

Ahogy azt említettük, amikor a RichNot2 programmal elkezdtünk dolgozni, a program legösszetettebb része a Fájl legördülő menü parancsainak – Új,

Megnyitás, Mentés, és Mentés másként – megvalósítása. Minden esetben követnünk kell, hogy az állomány megváltozott-e, és csak akkor elmenteni, ha szükséges. Ilyenkor felhasználót megkérdezhetjük, hogy el akarja-e menteni az állományt, mielőtt a program új állományt hoz létre, megnyit egy már létezőt, vagy kilép.

Ehhez felvettünk két új mezőt, és három metódust az alkalmazás formját leíró osztályba:

```
private
    FileName: string;
    Modified: Boolean;
  public
    function SaveChanges: Boolean;
    function Save: Boolean;
    function SaveAs: Boolean;
```

A FileName (állománynév) karakterlánc, és a Modified (módosított) jelző a form elkészítésekor kap értéket, és akkor változik meg, ha új állományt nyitunk meg, vagy a felhasználó átnevezi az állományt a Mentés másként paranccsal:

```
procedure TFormRichNote.FormCreate(Sender: TObject);
begin
  FileName := '';
  Modified := False;
end;
```

A jelző értéke azonnal megváltozik, amint új karaktereket gépelünk a RichEdit kontrolba, erre szolgál a következő OnChange eseménykezelő:

```
procedure TFormRichNote.RichEdit1Change(Sender:
TObject);
begin
  Modified := True;
end;
```

Amikor új állományt hozunk létre, a program megnézi, hogy a szöveg változott-e, és ha igen, akkor meghívja a SaveChanges (változások mentése) függvényt, ami megkérdezi a felhasználót, hogy el akarja-e menteni a változásokat, vagy sem, esetleg mégsem akar új állományt megnyitni.

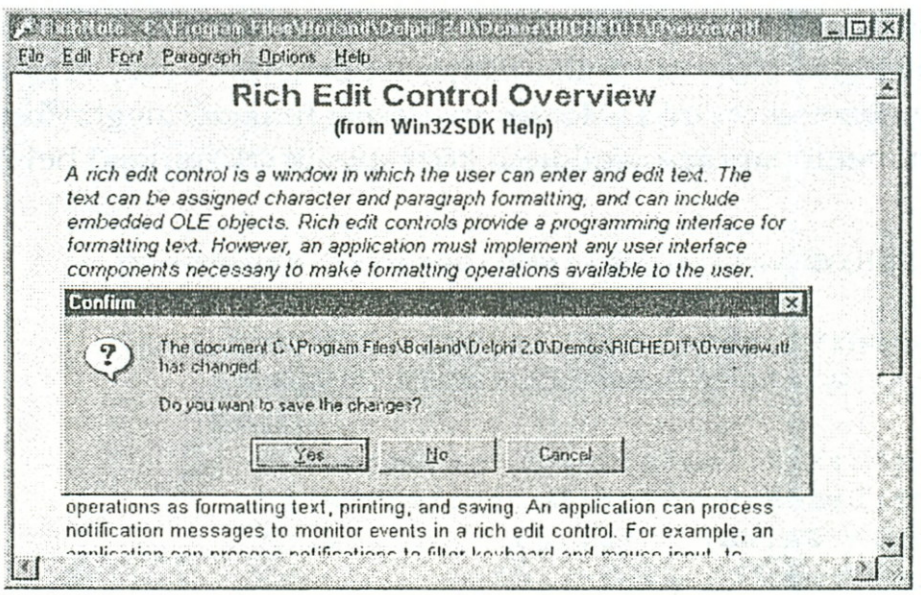
```
procedure TFormRichNote.New1Click(Sender: TObject);
begin
  if not Modified or SaveChanges then
  begin
    RichEdit1.Text := '';
    Modified := False;
    FileName := '';
    Caption := 'RichNote - [Untitled]';
  end;
end;
```

Ha a felhasználó az új állomány létrehozása mellett dönt, akkor néhány egyszerű dolgot kell tennünk, többek között a form címsorában „Untitled"-et (névtelent) kell használnunk az állomány neve helyett.

Rövid kiértékelés

Az „if not Modified or SaveChanges then" kifejezés némi magyarázatot igényel. Alapértelmezés szerint a Pascal komplex feltételek kiértékelésénél úgynevezett rövid kiértékelést alkalmaz. Az ötlet nagyon egyszerű: ha a „not Modified" kifejezés igaz, akkor biztosak lehetünk abban, hogy az egész kifejezés igaz lesz, ezért nem is kell kiértékelnünk a másodikat. Ebben az esetben a második kifejezés egy függvényhívás, ami csak akkor hívódik meg, ha a Modified igaz. Az `or` és `and` kifejezések ilyen kiértékelési módja megváltoztatható a **Complete Boolean Eval** (teljes feltétel kiértékelés) fordító direktíva beállítással. Ezt a Project Options dialógus **Compiler** oldalán találhatjuk meg.

A SaveChanges függvény által megjelenített, a 9.16. ábrán is látható dialógusablak három opciót kínál. Ha a felhasználó a Cancel (Mégse) gombot választja, a függvény hamis értékkel tér vissza. No (Nem) választása esetén semmi nem történik – az állomány nem kerül elmentésre-, de a függvény igaz értékkel tér vissza, azt jelezve, hogy, bár nem mentettük el az állományt, a kért művelet – például az új állomány létrehozása – elvégezhető. Ha a felhasználó Yes-t (Igen) választ, akkor az állományt elmentjük, és a függvény igazzal tér vissza.



9.16. ábra *A RichNot2 példa üzenetablaka, amikor a RichEdit kontrol szövege megváltozott, és nem mentettük el*

A függvény kódjában nézzük meg, hogy a MessageDlg (Üzenet dialógus) függvényhívást hogyan lehet „case" szerkezet értékeként használni:

```
function TFormRichNote.SaveChanges: Boolean;
begin
  case MessageDlg (
    'The document ' + filename + ' has changed.' +
    #13#13 + 'Do you want to save the changes?',
    mtConfirmation, mbYesNoCancel, 0) of
  idYes:
    // call Save and return its result
    Result := Save;
  idNo:
    // don't save and continue
    Result := True;
  else // idCancel:
    // don't save and abort operation
    Result := False;
  end;
end;
```

Tipp

A fenti MessageDlg hívásban explicit újsor karaktereket (#13) használtunk a kimenet olvashatóságának javítására. Egy másik megoldás lehet a Chr(13) hívás.

Hogy tényleg elmentsük az állományt, egy másik függvényt használtunk, ez a Save (Mentés). Ez a metódus elmenti az állományt, ha már van neve, vagy a felhasználótól kér egy nevet, a SaveAs (Mentés másként) funkció meghívásával. Látható még két, a menü parancsokkal nem közvetlenül összefüggő belső függvény is:

```
function TFormRichNote.Save: Boolean;
begin
  if Filename = '' then
    Result := SaveAs // ask for a file name
  else
  begin
    RichEdit1.Lines.SaveToFile (FileName);
    Modified := False;
    Result := True;
  end;
end;
```

```
function TFormRichNote.SaveAs: Boolean;
begin
  SaveDialog1.FileName := Filename;
  if SaveDialog1.Execute then
  begin
    Filename := SaveDialog1.FileName;
    Save;
    Caption := 'RichNote - ' + Filename;
    Result := True;
  end
  else
    Result := False;
end;
```

A megfelelő menükezelők helyett azért használtunk két függvényt a Mentés és Mentés másként műveletekre, mert arról is tudomást kellett szereznünk, ha a felhasználó megszakította a műveletet. Kétszer szereplő kód helyett, a Mentés, és a Mentés másként menüpontok is ezeket a függvényeket hívják, de itt a visszatérési értékeket figyelmen kívül hagyjuk:

```
procedure TFormRichNote.Save1Click(Sender: TObject);
begin
  if Modified then
    Save;
end;

procedure TFormRichNote.Saveas1Click(Sender: TObject);
begin
  SaveAs;
end;
```

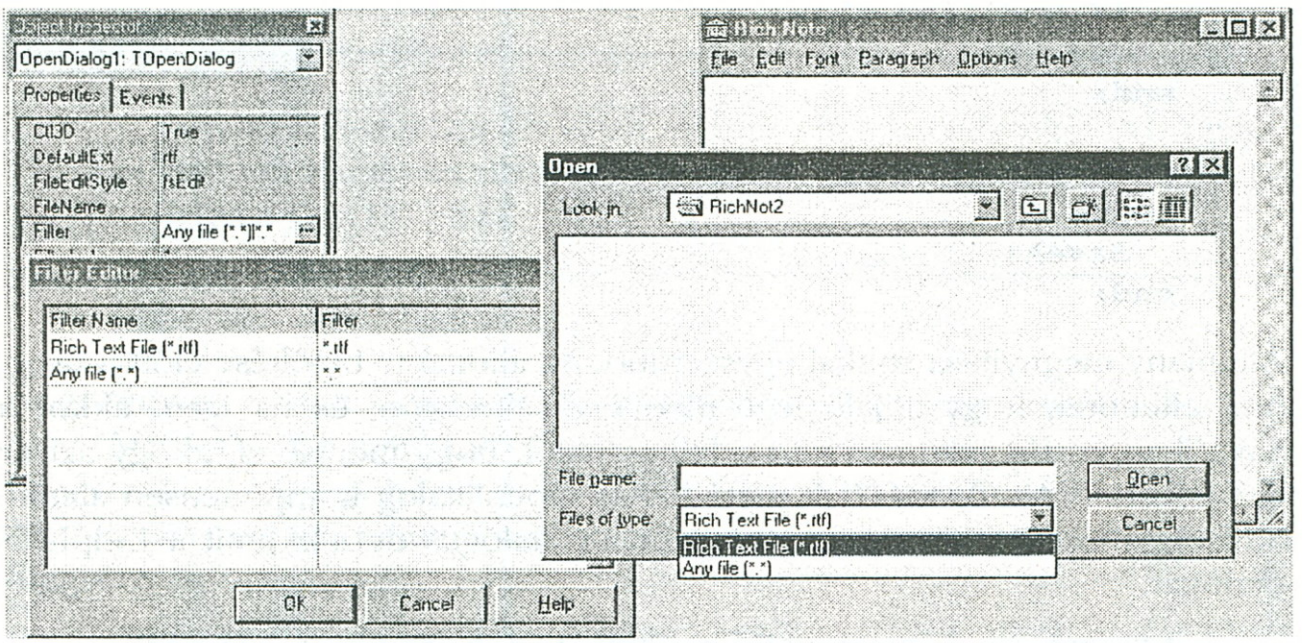
Állomány megnyitása sokkal egyszerűbb. Az állomány betöltése előtt a program ellenőrzi, hogy a jelenlegi állomány változott-e, és ha igen, akkor a SaveChanges függvénnyel kéri a felhasználót, hogy mentse el, ahogy azt az előbb láttuk. Az Open1Click metódus az OpenDialog komponensen alapul, ami egy másik, a Windows által definiált dialógus doboz, amit a Delphi is támogat:

```
procedure TFormRichNote.Open1Click(Sender: TObject);
begin
  if not Modified or SaveChanges then
    if OpenDialog1.Execute then
    begin
      Filename := OpenDialog1.FileName;
      RichEdit1.Lines.LoadFromFile (FileName);
      Modified := False;
      Caption := 'RichNote - ' + FileName;
    end;
end;
```

Az állomány műveletekhez kapcsolódik még az is, hogy a NotesForm OpenDialog, és SaveDialog komponenseinek ugyanazok a Filter (szűrő), és DefaultExt (alapértelmezett kiterjesztés) tulajdonságai, ahogy az a következő szöveges leírás részletből látszik:

```
object OpenDialog1: TOpenDialog
  DefaultExt = 'rtf'
  FileEditStyle = fsEdit
  Filter = 'Rich Text File (*.rtf)|*.rtf|Any file
    (*.*)|*.*'
  Options = [ofHideReadOnly, ofPathMustExist,
    ofFileMustExist]
end
```

A Filter tulajdonság karakterlánca – amit egy sorba kell írni – két párt tartalmaz, amiket a | szimbólum választ el. A párok egy, a File Open és a File Save dialógusok típus mezőjében megjelenő szöveges leírásból, és a mappa állományaira alkalmazandó szűrőből áll, mint amilyen a *.RTF. Szűrők beállításához meghívhatjuk ennek a tulajdonságnak a szerkesztőjét, ami egy két oszlopot tartalmazó listát mutat, ahogy az a 9.17. ábrán is látható.



9.17. ábra *A RichNot2 program szabványos Open File dialógusablaka. Baloldalt láthatjuk az Object Inspector és a Filter Editor segítségével beállított Filter tulajdonság értékeit*

Az állományokkal kapcsolatos fenti metódusokat még az OnCloseQuery eseményt kezelő FormCloseQuery (form lezárási kérelem) metódusból is használjuk, ami minden alkalommal meghívódik, amikor a felhasználó megpróbálja bezárni a formot, vagy ki akar lépni a programból. Ezt többféleképpen

megtehetjük, duplán kattinthatunk a rendszer menü ikonján, kiválaszthatjuk a rendszermenü Close parancsát, megnyomhatjuk az ALT+F4 kombinációt, vagy meghívhatjuk a kód Close metódusát, ahogy az a Fájl menü Kilépés pontjára történik.

A FormCloseQuery metódusban a hivatkozás szerint átadott CanClose (bezárható) paraméter beállításával eldönthetjük, hogy a programot le lehet-e állítani. Itt is, ha az állomány módosult, akkor meghívjuk a SaveChanges függvényt, és a visszatérési értékét ismét a rövid kiértékelési technikával használjuk:

```
procedure TFormRichNote.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  CanClose := not Modified or SaveChanges;
end;
```

A Fájl menü utolsó menüpontja a Nyomtatás parancs. A nyomtatásról részletesen a 28. Fejezetben lesz szó, de mivel a RichEdit komponensnek nagyon egyszerűen kezelhető nyomtatási szolgáltatása van, ezért most mégis elkészül ez a menüpont. A következő kód nagyon szép nyomtatási képet eredményez:

```
procedure TFormRichNote.Print1Click(Sender: TObject);
begin
  RichEdit1.Print (FileName);
end;
```

A Bekezdés menü

A Fájl menühöz képest a többi legördülő menü egyszerűbb lesz. A Bekezdés menü kódja az elemeinek tulajdonságaira épül; íme a szöveges leírása, amit a DFM állományból másoltam ki:

```
object Paragraph1: TMenuItem
      Caption = '&Paragraph'
      object LeftAligned1: TMenuItem
        Caption = '&Left Aligned'
        Checked = True
        GroupIndex = 1
        RadioItem = True
        OnClick = RightAligned1Click
      end
      object RightAligned1: TMenuItem
        Caption = '&Right Aligned'
        GroupIndex = 1
        RadioItem = True
        OnClick = RightAligned1Click
      end
      object Centered1: TMenuItem
        Caption = '&Centered'
```

```
            GroupIndex = 1
            RadioItem = True
            OnClick = RightAligned1Click
          end
        end
```

Amint az látható, a program rádió menüpontokat használ, azáltal, hogy a három elem GroupIndex tulajdonságát ugyanarra az értékre, a RadioItem tulajdonságát pedig igazra állítja. A menüpontok az OnClick eseményüknél ugyanazt a RightAligned1Click metódust használják. Lent látható a metódus kódja, ami a menüpont menübeli helyén – amit a MenuIndex tulajdonság mutat-, és a TAlignment típus értéksorrendjén alapul. Ez egy kis trükk, de jól működik.

```
procedure TFormRichNote.RightAligned1Click(Sender:
TObject);
begin
  RichEdit1.Paragraph.Alignment :=
    TAlignment ((Sender as TMenuItem).MenuIndex);
  (Sender as TMenuItem).Checked := True;
end;
```

Az eljárás először beállítja a kiválasztott, vagy a szerkesztő kurzor által mutatott bekezdés igazítási stílusát, azután bekapcsolja az aktuális menüpontot – vagyis azt, amelyik a metódust meghívta, a Sender objektumot. Amint látható, a kódban van néhány irányított típuskonverzió, az „as" kulcsszó segítségével, ez az, amit meg kell tennünk, ha általános kódot szeretnénk írni – például olyankor, ha ugyanazt a metódust különböző komponensek eseményeihez akarjuk kapcsolni.

Vegyük észre, hogy a menüpont kijelzése csak addig helyes, ameddig meg nem változtatjuk a kijelölést, vagy arrébb nem megyünk a kurzorral. Emiatt, a RichEdit komponens OnSelectionChange metódusát használjuk a menü jelölés aktualizálására:

```
procedure TFormRichNote.RichEdit1SelectionChange(Sender:
TObject);
begin
  Paragraph1.Items [Integer
(RichEdit1.Paragraph.Alignment)].
    Checked := True;
end;
```

A Betűtípus menü

A Betűtípus menü is az előző koncepcióra épül, de ebben két csoport rádió menüpont van, és néhány egyéb pont, hagyományos pipa jelöléssel. Mindkét csoport egyetlen eseménykezelőt használ, de más megközelítéssel kezeli le az elem kiválasztását. A tulajdonságok részletes listáját a CD-n lévő

RichForm.DFM állományban nézhetjük meg, mert ez a lista túl hosszú ahhoz, hogy ide bekerüljön.

Az első csoport – a betűtípus kiválasztására szolgáló pontok- kódja a következő egyszerű trükkön alapul: a kiválasztott betűtípus neve a bevezető & karakter kivételével megegyezik a menüpont nevével:

```
procedure TFormRichNote.TimesRoman1Click(Sender:
  TObject);
var
  FontName: string;
begin
  // get the font name and remove the &
  FontName := (Sender as TMenuItem).Caption;
  Delete (FontName, 1, 1);
  // change selected text font
  if RichEdit1.SelLength > 0 then
    RichEdit1.SelAttributes.Name := FontName;
  (Sender as TMenuItem).Checked := True;
end;
```

Ez a kód a kiválasztott részt változtatja meg a RichEdit1 komponens SelAttributes (kiválasztás tulajdonságai) tulajdonság használatával, ahogy azt a RichNote példa tette a nyolcadik fejezetben. Figyeljük meg, hogy milyen egyszerű ezt a kódot kibővíteni új menüpontokkal, ha betartjuk azt, hogy a menüpont neve a font neve elé tett & karakter lesz. Ezután már csak a GroupIndex tulajdonságát kell a többi pontéval megegyezőre állítani – jelen esetben 1-re – és ha a RadioItem tulajdonságot igazra állítjuk, valamint az OnClick eseményt a TimesRoman1Click metódusra irányítjuk, akkor éppen úgy fog viselkedni, mint a már meglévő betűtípus-választó elemek.

A második kontrol csoport a félkövér és dőlt stílusokat állítja be. Ezt két hasonló, de különálló metódus végzi, ezek közül az egyik kódja következik:

```
procedure TFormRichNote.Bold1Click(Sender: TObject);
begin
  Bold1.Checked := not Bold1.Checked;
  if RichEdit1.SelLength > 0 then
    with RichEdit1.SelAttributes do
      if Bold1.Checked then
        Style := Style + [fsBold]
      else
        Style := Style - [fsBold];
end;
```

A menü utolsó részében van egy másik rádiógomb csoport, ami a betűméretet állítja. Ezeknek a pontoknak a neve megegyezik a hivatkozott fontmérettel, és ugyanez a szám szerepel a Tag tulajdonságukban, ami a kód elkészítését

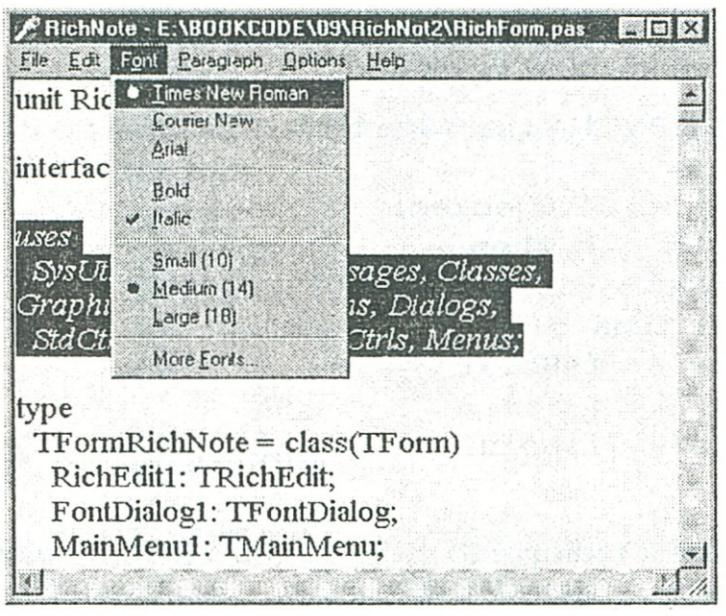
nagyon leegyszerűsíti. A három menüpontot itt is egyetlen metódus kezeli le, és új pontokat is nagyon könnyen adhatunk a csoporthoz.

```
procedure TFormRichNote.Large1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
    RichEdit1.SelAttributes.Size :=(Sender as
TMenuItem).Tag;
  (Sender as TMenuItem).Checked := True;
end;
```

A menü utolsó pontja meghívja a betűtípus dialógust. Vegyük észre, hogy a dialógusablak által visszaadott betűtípust közvetlenül nem adhatjuk értékül a SelAttributes tulajdonságnak, ehhez meg kell hívnunk az Assign (hozzárendelés) metódust:

```
procedure TFormRichNote.More1Click(Sender: TObject);
begin
  FontDialog1.Font := RichEdit1.Font;
  if FontDialog1.Execute then
    RichEdit1.SelAttributes.Assign (FontDialog1.Font);
  // update the check marks
  RichEdit1SelectionChange (Self);
end;
```

Az összes eddigi menüpont, ami az aktuális kiválasztott rész tulajdonságait módosította, helyesen beállította a jelöléseket, és a rádió menüpontokat, ahogy az a 9.18. ábrán látható. A fenti metódus e helyett a RichEdit1SelectionChange (RichEdit1 kiválasztás változtatás) metódust hívja meg, ami a RichEdit komponens OnSelectionChange eseményének kezelője.



9.18. ábra *A jelzések és a rádió menüpontok a RichNot2 példa Betűtípus menüjében a jelenlegi beállításokat tükrözik*

Ez az eljárás végignézi a csoportokat, hogy eldöntse, mely elemeket kell bejelölni. Az előző példákhoz hasonlóan ez a kód is úgy készült, hogy könnyen bővíthető legyen új menüpontokkal – a bekezdés igazításának beállítására a lista a korábban már látott kódot tartalmazza:

```
procedure TFormRichNote.RichEdit1SelectionChange(Sender:
TObject);
var
  FontName: string;
  I: Integer;
begin
  // check the font name radio menu item
  FontName := '&' + RichEdit1.SelAttributes.Name;
  for I := 0 to 2 do
    with Font1.Items [I] do
      if FontName = Caption then
        Checked := True;
  // check the bold and italic items
  Italic1.Checked :=
    fsItalic in RichEdit1.SelAttributes.Style;
  Bold1.Checked :=
    fsBold in RichEdit1.SelAttributes.Style;
  // check the font size
  for I := Small1.MenuIndex to Large1.MenuIndex do
    with Font1.Items [I] do
      if Tag = RichEdit1.SelAttributes.Size then
        Checked := True;
  // check the paragraph style
  Paragraph1.Items [Integer
    (RichEdit1.Paragraph.Alignment)].Checked := True;
end;
```

Ez az eljárás nem működik tökéletesen. Ha olyan fontot választunk, ami nem szerepel a menüben, a kiválasztás nem lesz helyes, mert az előző marad aktív. Ehhez vagy el kell távolítanunk minden jelölést, ha a kiválasztott nem található meg, vagy új menüpontot kell felvennünk erre a különleges esetre, de a példa így is elég összetett, így ez a kis hiba akár benne is maradhat.

A Beállítások menü

A RichNot2 példa utolsó legördülő menüpontja a Beállítások menü. Ennek három, egymáshoz nem kapcsolódó menüpontja van, amivel a felhasználói felületet módosíthatjuk, a szöveg hosszát állapíthatjuk meg és írathatjuk ki. Az első parancs a szín választó dialógust jeleníti meg, amivel a RichEdit komponens háttérszínét állíthatjuk be:

```
procedure TFormRichNote.BackColor1Click(Sender:TObject);
begin
  ColorDialog1.Color := RichEdit1.Color;
  if ColorDialog1.Execute then
    RichEdit1.Color := ColorDialog1.Color;
end;
```

A második paranccsal a szöveget csak olvashatónak állíthatjuk be. Elméletben ez a tulajdonság a lemezen lévő állomány állapotából adódna, de ebben a példában a felhasználóra bízzuk az átállítást:

```
procedure TFormRichNote.ReadOnly1Click(Sender: TObject);
begin
  RichEdit1.ReadOnly := not RichEdit1.ReadOnly;
  ReadOnly1.Checked := not ReadOnly1.Checked;
end;
```

Az utolsó menüpont egy metódust hív meg, ami megszámolja a szövegben lévő karaktereket, és az összeget kiírja egy üzenet ablakban. A metódus magját a RichEdit kontrol GetTextLen (szöveghossz megállapítás) függvényének meghívása alkotja, aminek visszaadott értékét jelenítjük meg:

```
procedure TFormRichNote.Countchars1Click(Sender:
  TObject);
begin
  MessageDlg (Format (
    'The text has %d characters',
    [RichEdit1.GetTextLen]),
    mtInformation, [mbOK], 0);
end;
```

A Beállítások menü utolsó pontjának OnClick eseménykezelőjével véget ér a példa. Ahogy azt az elején említettük, elég hosszú és összetett, de a cél az volt, hogy bemutassuk, hogyan lehet egy valódi alkalmazás menüjét elkészíteni. A Fájl menü pontjait azért vettük ilyen részletesen, mert ezt szinte minden állományokkal kapcsolatos alkalmazásban el kell készítenünk.

Most már áttérhetünk a menük egy másik részére, a jobb egérgombbal előhívható helyi menükre.

Előugró menük

Windows 95-ben megszokott dolog, hogy az alkalmazásoknak speciális, jobb egérgombra megjelenő helyi menüik vannak. Az így megjelenő – a Windows terminológiában előugró menünek nevezett – menü helye általában a kattintás helyétől függ. Az ilyen menüket könnyű használni, hiszen csak néhány, az éppen kiválasztott elemhez kapcsolódó parancsot tartalmaznak, és gyorsabban is kezelhetők, mint a nagy menük, hiszen nem kell mozgatnunk az egeret a menüsorig és vissza.

Delphiben alapvetően két megoldás kínálkozik előugró menük készítésére a megfelelő komponens használatával. A menük kezelését ráhagyhatjuk a Delphire, vagy választhatjuk azt is, hogy erről mi gondoskodunk. Most mindkét utat bemutatjuk, az egyszerűbb elsővel kezdve.

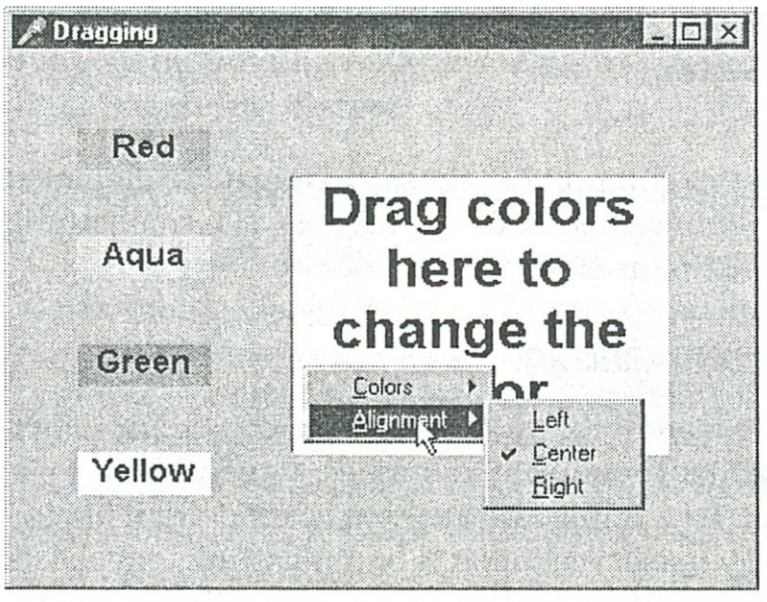
Előugró menü készítéséhez csak néhány alapvető műveletet kell elvégeznünk. Készítsünk egy PopupMenu (előugró menü) komponenst, adjunk hozzá néhány elemet, a form PopupMenu tulajdonságánál állítsuk be ezt a komponenst, és készen is vagyunk. Természetesen a menü pontjaihoz tartozó kezelőket ugyanúgy meg kell írnunk, mint a hagyományos menüknél.

Automatikus helyi menü

Lássunk most egy példát a helyi menük elkészítésére, ami a nyolcadik fejezetben szereplő Dragging példa kibővítése, és a Local1 nevet kapta. Először adjuk hozzá az első PopupMenu komponenst a formhoz, és kapcsoljuk is hozzá a form PopupMenu tulajdonságával. Amikor ez megvan, futtassuk a programot, és ha jobb gombbal kattintunk, a form megjeleníti a helyi menüt.

Hozzáadhatunk még egy második előugró menü komponenst is, aminek két szintje van, de ezt a jobb oldali, LabelTarget nevű StaticText komponenshez kapcsoljuk; ehhez a komponens PopupMenu tulajdonságát kell beállítanunk. Az eredmény a 9.19. ábrán látható. Az első csoportba tartozó, a Colors legördülő menüvel kapcsolatos négy metódus csak egy-egy színt választ ki:

```
procedure TDraggingForm.Aqua1Click(Sender: TObject);
begin
  LabelTarget.Color := clAqua;
end;
```



9.19. ábra *A Local1 példa második előugró menüjének bejelölt Igazítás legördülő menüje*

Tipp

Ahogy azt a hetedik fejezetben megbeszéltük, a form komponensei gyakran a form tulajdonságait használják Ezt különleges tulajdonságok jelölik, mint a ParentColor (szülő szín) vagy ParentFont (szülő betűtípus). Ha ezeket a tulajdonságokat igazra állítjuk, a komponens saját tulajdonságait a rendszer figyelmen kívül hagyja, és a form értékeit használja helyettük. Ez általában nem probléma, mert amint a komponens valamelyik tulajdonságát – például a betűtípust – átállítjuk, az ezzel kapcsolatos szülő érték – ParentFont – automatikusan hamisra vált.

A Transparent (átlátszó) parancs a szülő form színét választja ki aktuális színnek, a ParentColor (szülőszín) tulajdonság igazra állításával.

A legördülő menü utolsó pontja, az User Defined (felhasználó által definiált), a szabványos Color Selection (szín választó) ablakot kínálja fel. Az előugró menü második legördülő menüjébe tartozó három parancs a nagy címke igazítási módját állítja, a megfelelő választást kipipálja, a többinek megszünteti a kijelöltségét.

Íme a három metódus egyike:

```
procedure TDraggingForm.Center1Click(Sender: TObject);
begin
  LabelTarget.Alignment := taCenter;
  Left1.Checked := False;
  Center1.Checked := True;
  Right1.Checked := False;
end;
```

Az előugró menüben használhatjuk a főmenü összes sajátosságát, az elemeket megjelölhetjük, kikapcsolhatjuk, eltüntethetjük és így tovább. Használhattunk volna rádió menüelemeket is, de ez nem elterjedt.

Előugró menü módosítása aktiváláskor

Miért ne használhatnánk ugyanezt a technikát a színek megjelölésére is? Lehet, de ez nem jó megoldás. Egyrészt hat menüpont van amire figyelni kell, másrészt a szín akkor is megváltozhat, ha a felhasználó egy másik baloldali címkéről ráhúzza azt. Emiatt, és hogy egy másik technikát is bemutassunk, egy másik megközelítést választottunk.

Amikor egy előugró menü megjelenik, az alkalmazásunk kap egy OnPopup eseményt. Az ehhez kapcsolódó metódusban elhelyezhetjük az éppen aktuális szín mellé a pipát, a beállítási módtól függetlenül:

```
procedure TDraggingForm.PopupMenu2Popup(Sender:
  TObject);
var
  I: Integer;
begin
  {unchecks all menu items
    (not required for radio menu items)}
  with Colors1 do
    for I := 0 to Count - 1 do
      Items[I].Checked := False;

  {checks the proper item}
  case LabelTarget.Color of
    clRed: Red1.Checked := True;
    clAqua: Aqua1.Checked := True;
    clGreen: Green1.Checked := True;
    clYellow: Yellow1.Checked := True;
  else
    if LabelTarget.ParentColor then
      Transparent1.Checked := True
    else
      UserDefined1.Checked := True;
  end;
end;
```

A metódus kódja igényel némi magyarázatot. Kezdetben az összes menüpont jelölését eltávolítjuk egy „for" ciklussal, ami a Colors1 menü Items tömbjén megy végig. Ennek a ciklusnak az az előnye, hogy a menüpontok számától függetlenül mindig működik. Ezután a program egy „case" kifejezéssel megjelöli a megfelelő elemet.

Előugró menük kezelése programból

A Local1 programban láthattuk az automatikus előugró menük használatát. Másik megoldás lehet, ha az AutoPopup (magától előugrik) tulajdonságot hamisra állítjuk, vagy az előugró menüt nem kötjük sehova, hanem a komponens Popup (előugrik) metódusával jelenítjük meg a képernyőn. Ez az eljárás két paramétert igényel, ezek a menü helyének x és y koordinátái, amivel az a probléma, hogy ezek a pont képernyő-koordinátái, és nem kliens koordináták, vagyis nem a form kliens területéhez viszonyítottak.

Példaként egy már létező menüs alkalmazást vettünk – az ebben a fejezetben leírt MenuOne példa harmadik változatát – és hozzáadtunk egy különleges előugró menüt. Az ötlet az volt, hogy van két előugró menü, egy, amelyik a

színeket állítja, és egy, amelyik a szöveg igazítását, és amikor a felhasználó jobb gombbal kattint a címkén, ezek közül az egyik megjelenik. Valódi alkalmazásokban általában valami alapján el kell döntenünk, hogy melyik menüt jelenítsük meg, itt azonban egy egyszerű, és sajátos szabályt követtünk: amikor a felhasználó jobb gombot nyom, a menü megváltozik. Ezzel csak annyi a célunk, hogy megmutassuk, hogyan tehetjük meg ezt a legegyszerűbben.

A két előugró menü nagyon egyszerű, és a szükséges eseménykezelőik is megvannak már a főmenüben. Ezeket az előugró menüket úgy is elkészíthetjük, hogy a főmenüt egy menü sablonba másoljuk, és azt használjuk alapul. Az egyetlen szükséges változtatás az, hogy el kell távolítanunk belőle a gyorsbillentyűket.

Amikor a felhasználó jobb gombbal kattint a címkére, ami a form egész felületét eltakarja, akkor a metódus megjeleníti a két menü egyikét. Az OnClick esemény helyett most az OnMouseDown eseményt használjuk, mert ez a kattintás koordinátáit is átadja paraméterként.

A koordináták a címkéhez viszonyítottak, ezért át kell alakítanunk képernyő koordinátákká a címke ClientToScreen (kliensből képernyővé) metódusával:

```
procedure TFormColorText.Label1MouseDown(Sender:
  TObject; Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
var
  ClientPoint, ScreenPoint: TPoint;
begin
  if Button = mbRight then
  begin
    ClientPoint.X := X;
    ClientPoint.Y := Y;
    ScreenPoint := Label1.ClientToScreen (ClientPoint);
    Inc (ClickCount);
    if Odd (ClickCount) then
      PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
    else
      PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
  end;
end;
```

Ebben az eljárásban először ellenőriznünk kell, hogy jobb gombbal kattintottunk-e, ezután a kattintás koordinátáit kliens koordinátákból képernyő koordinátákká alakítjuk, mert a PopupMenu komponens Popup metódusának képernyő koordináták kellenek.

Az utolsó teendőnk, hogy a második előugró menü pontjainál beállítsuk a megfejelő jelöléseket. Ehhez a főmenü jelöléseit másoljuk le, még a menü megjelenítése előtt.

```
if Odd (ClickCount) then
  PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
else
begin
  {set the check marks as in the main menu}
  Left2.Checked := Left1.Checked;
  Center2.Checked := Center1.Checked;
  Right2.Checked := Right1.Checked;
  PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
end;
```

Másik megoldás lehet – és a Local2 példában ez is szerepel-, ha a jelöléseket a főmenüvel egyszerre állítjuk, mint a lenti kódban:

```
procedure TFormColorText.Left1Click(Sender: TObject);
begin
  Label1.Alignment := taLeftJustify;
  Left1.Checked := True;
  Left2.Checked := True;
end;
```

Általánosabb alkalmazásban igény lehet egy általánosabb rutinra, ami következetesebben állítja a jeleket, de ebben a példában ez a két egyszerű technika is megfelel.

Hogyan tovább?

Ebben a fejezetben láthattunk, hogy hogyan készíthetünk főmenüt és előugró menüket Delphiben. Megbeszéltük a szabványos iránymutatásokat a legördülő menük, és menüpontok neveire, a gyorsbillentyűket, a jelöléseket, a grafikus, és a helyi menüket, valamint sok más témát is. Innen többfelé elindulhatunk, például készíthetünk menüket dinamikusan (futásidőben), vagy menük részeit másik menükbe másolhatjuk, ahogy azt a SysMenu példában láttuk. Menükkel foglalkozik még a tizenharmadik, és a tizenötödik fejezet, ahol a menük egyesítését nézzük meg. A SysMenu példával újra találkozhatunk a huszonötödik fejezetben, menüket pedig szinte mindegyik fejezetben fogunk használni.

A következő lépés azonban az, hogy megismerjük a Delphi programozás egyik legáltalánosabb elemét, a formot. Már eddig is sokszor használtunk formokat, de még maradt sok új, és fontos dolog, amiről beszélnünk kell. Ezután áttérünk a grafikus komponensekre, az eszköztárakra, státuszsorokra, párbeszédablakokra, és így tovább.

10. FEJEZET

A formok mélyebb megismerése

E fejezet tartalma:

- Fontok és ablakok
- Formstílusok beállítása
- Keretstílus
- A form helyzetének és méretének állítása
- Formok készítése automatikusan
- A billentyűzet-események figyelése
- Egéresemények figyelése
- Rajzolás a formra
- Rajzolás és festés a Windows-ban
- A Delphi kimeneti komponensei

A formok mélyebb megismerése

A könyv e pontjáig eljutva valószínűleg saját magunk is használni tudjuk a Delphi alapvető komponenseit és a menüket. Most vizsgáljuk meg alaposan a Delphi-fejlesztés egyik központi elemét: a formot. Már egészen a kezdetektől használjuk a formokat, de részletesen nem tárgyaltuk a hozzájuk kapcsolódó programozási technikákat, a használható tulajdonságokat és metódusokat.

E fejezet főleg a formok stílusához, méretéhez és elhelyezkedéséhez tartozó dolgokat ismerteti, majd szentelünk egy kis időt a formra, mint a kimenet és bemenet egy eszközére, végül ismertetem a formok kifestésének módjait. Bizonyos dolgokat nem árt előre tisztázni, ezért egy kis elméleti áttekintéssel kezdeném.

Formok és ablakok

Emlékszünk a könyv első fejezetére? „A form egy ablak..." Már az elején utaltam arra, hogy van valamiféle összefüggés a Delphiben készített formok és a Windows rendszer ablakai között. Itt az ideje, hogy mélyebben megvizsgáljuk e kérdést.

A Windows felhasználói felületét túlnyomó részt ablakok alkotják. Ebből következik, hogy a Delphi komponensei is ablak-alapúak – bár nem mindegyik az. A különbség nem nyilvánvaló, a felhasználónak igazán nem is tűnik fel, de egy programozónak érdemes megjegyezni az alábbi definíciókat:

- Felhasználói szemszögből egy ablak a rendszer olyan, kerettel, címsorral ellátott eleme, mely mozgatható, lezárható, bizonyos esetekben maximalizálható és minimalizálható. Az ablakok a képernyőn vagy egy másik ablakon belől mozgathatók (ezek az MDI alkalmazások ablakai). A felhasználó két csoportra oszthatja az ablakokat: „normál" ablakokra és párbeszédablakokra.

- Technikailag az ablak egy memóriabeli adatterület, mely gyakran a képernyő egy vizuális eleméhez kapcsolódik, és tartozik hozzá valamiféle futtatható kód. A Windows rendszer egy listában tartja nyilván a létrehozott ablakokat, és mindegyikhez egyedi azonosítót, ún. *window handle*-t rendel. Ezek egy része megfelel a klasszikus ablak-fogalomnak (ezt ismertettem az előbbi definícióban), más részük vizuális komponensként jelenik meg, egyeseket a rendszer ideiglenesen hoz létre például egy legördülő menü megjelenítéséhez, míg másokat az alkalmazások készítenek, de a felhasználó nem látja őket.

Minden ablakban van valami közös: nyilvántartja őket a rendszer és van egy különleges kódrészletük, mely az ablakhoz kapcsolódó üzenetek kezelését végzi, ezt hívják ablakkezelő függvénynek (window procedure).

A Delphi az alacsony szintű rendszerüzeneteket eseményekké alakítja, de ha szükség van rá, saját magunk is lekezelhetjük a Windows üzeneteit. A Delphi segítségével azonban nem kell nagyon belemélyednünk az események kezelésébe, figyelmünket az alkalmazás-fejlesztés lényegi részére fordíthatjuk.

Megjegyzés

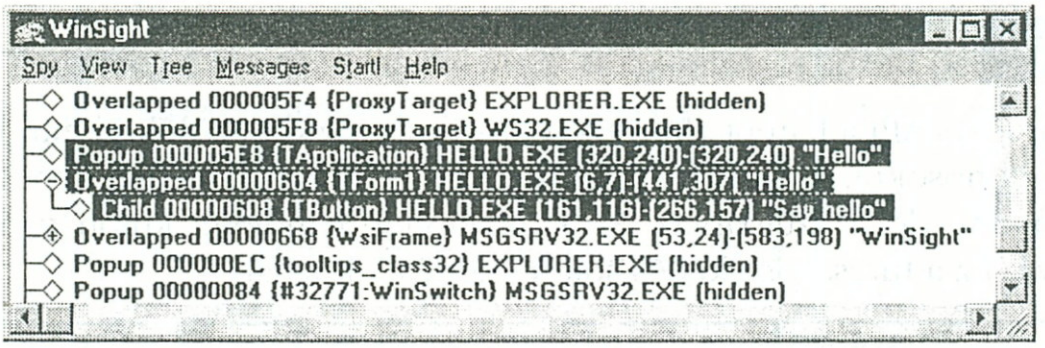
A Windows ablakok részére fenntartott memóriaterülete véges. Túl sok ablak létrehozása csökkenti az ún. rendszer-erőforrásokat (system resources). A Windows 3.1 igen korlátozott volt e téren, a Windows 95-ben ez a határ jelentősen ki lett tolva, míg Windows NT-ben gyakorlatilag nem is létezik. Ha túl sok ablak lett létrehozva a rendszerben (ideértve a kontrolokat és a rejtett ablakokat is!), nem lehet új ablakot készíteni, így alkalmazásunk valószínűleg nem lesz képes normálisan működni. Ez az egyik ok, de annak, hogy a Delphi nem ablakként kezeli például a Label komponenseket, holott a Windows ezeket eredetileg ablakként implementálja. Így jelentősen csökkenthető az alkalmazások memóriahasználata.

Eme definíciókkal felvértezve próbáljuk megérteni a formok szerepét. Felhasználói szemszögből a formok „normál ablakokat", MDI gyermekablakokat és párbeszédablakokat jelentenek. Viselkedésük főleg a hozzájuk tartozó kódtól függ, de néhány nagyon fontos tulajdonság is befolyásolja megjelenésüket. Számos egyéb komponens is ablakokon alapul, de ezeket csak a második, technikai megfogalmazás alapján tekintjük ablaknak.

Egy egyszerű példa: az első fejezet Hello programját vizsgáljuk meg a Delphivel kapott WinSight alkalmazáson keresztül. Láthatjuk a rendszerben létező ablakokat, ezek között megtaláljuk természetesen a mi alkalmazásunk ablakait is (10.1. ábra). A következők láthatjuk:

- Magát a formot, mint főablakot, Hello címsorral. Ez TForm1 osztály példánya, egy átlapolt (overlapped) ablak.
- Egy TButton osztályba tartozó gyermekablakot (child window), mely a Say Hello gombnak felel meg.
- Egy rejtett ablakot, szintén Hello felirattal. Ez az ún. alkalmazás-ablak (application window), mely a TApplication osztályból származik.

Megfigyelhető, hogy a kapcsos zárójelekben a nevek a Delphi komponensek osztályainak felelnek meg.



10.1. ábra *A Hello alkalmazás ablakai a WinSight alkalmazásban*

Átlapolt, felbukkanó és gyermekablakok

A különböző ablakok szerepének megértéséhez szükséges tisztában lennünk a Windows néhány technikai részletével. Bár ezek nem egyszerű fogalmak, érdemes megismerni őket.

Minden ablak, amit létrehozunk, három fő ablakstílus egyikébe fog tartozni. E stílusok:

- Az *átlapolt (overlapped)* ablakok mindenki számára ismerősek, ezek testesítik meg a klasszikus ablakokat.
- A *felbukkanó (pop-up)* ablakok főképp párbeszédablakok és üzenetablakok készítésekor használatosak. Jelenlétüknek történelmi oka van: a Windows korai verzióiban a normál ablakok nem fedhették egymás, és csak a felbukkanó ablakok takarhattak el bizonyos részt a többi ablakból. Napjaink Windowsában a pop-up ablakok nagyon hasonlítanak az átlapolt ablakokra.
- A harmadik csoportot a *gyermekablakok (child windows)* alkotják. Eredetileg a formokon lévő kontrolok tartoztak ide, de olyan ablakok létrehozására is használhatjuk e stílust, melyek nem hagyhatják el szülőablakuk területét. MDI gyermekablakok is létrehozhatók ily módon, de ahogy a 15. fejezetben látni fogjuk, erre más technika is létezik.

Fontos megjegyezni, hogy csak a gyermekablakoknak lehet *szülőjük (parent window)*, az összes többi ablaknak tulajdonosa van (owner). A tulajdonos is egy ablak, mely állandóan üzeneteket vált a hozzá tartozó ablakokkal, értesül róla, ha azokkal valami történik (például minimalizálják). Általában a tulajdonos egyben a szülő is, és gyermekablakait arra kényszeríti, hogy kliensterületén maradjanak. A gyermekablakok ezért nem is képernyő-koordinátákat, hanem a szülőjük kliensterületének koordinátáit használják.

Figyelem!

A Windows API a Parent elnevezést használja mind a szülőkre, mind a tulajdonosokra. A GetParent API függvény is mindkét elemet képes visszaadni, de a szülőablak és a tulajdonos handle-je külön van tárolva. Ez a furcsa viselkedés sok félreértést okozhat.

Delphiben minden form, még a párbeszédablakok is átlapolt ablakként jönnek létre, az ablakalapú komponensek (a kontrolok) tulajdonosa pedig a form, amin elhelyezzük őket. Szülőjük viszont nem csak a form lehet, hanem egy ún. *konténer (container)* komponens is, például egy GroupBox vagy egy Panel. Ha egy rádiógombot egy GroupBox komponensre helyezünk, a GroupBox lesz a szülő, viszont a form lesz a tulajdonos. A felbukkanó ablakokat a Delphi csak a rejtett alkalmazás-ablak létrehozásához használja, a rendszer azonban ilyenekre építi párbeszéd- és üzenetablakait.

Az alkalmazás is ablak

Ha figyelmesen vizsgáljuk a WinSight által nyújtott információt, furcsa dolgot vehetünk észre: az alkalmazás maga is egy ablak! A program főablaka teljesen úgy viselkedik, mintha ő lenne az alkalmazás központi része, valójában e formnak is van tulajdonosa, a rejtett alkalmazás-ablak. Ez az ablak csak akkor „jelenik meg", ha kis méretre csukjuk az alkalmazást.

Megjegyzés

Már korábban is láthattuk, hogy a minimalizált ablak Tálcán előhívott rendszermenüje eltér a normál méretű ablakétól, és bizonyos esetekben még címsoruk is különbözhet.

Akkor hát mi a szerepe e rejtett ablaknak? Elsősorban inicializációs feladatokat lát el, de kódjának segítségével (**View | Project Source**) eldönthetjük, hogy milyen sorrendben jöjjenek létre formjaink.

Ez az Application objektumhoz tartozó alkalmazás-ablak fogja össze a program összes ablakát. Az egységességnek ez a fajtája alapvető például olyankor, amikor kiválasztjuk éppen nem aktív alkalmazásunk egyik ablakát. Ilyenkor – az alkalmazás-ablaknak köszönhetően – a programunk összes ablaka előtérbe kerül, nem csak az az egy, melyen kattintottunk.

Az Application objektum egyike azoknak a komponenseknek, mely nem található meg a komponens-palettán. Tulajdonságai között például megtalálhatjuk az alkalmazás futtatható állományának elérési útját és a program címsorát. E Title tulajdonság alapértéke általában a futtatható fájl neve kiterjesztés nélkül, ez jelenik meg a képernyőn, ha az ALT-TAB billentyűkkel váltogatunk a futó alkalmazások között. Mivel az alkalmazás-ablak és a főform Title tulajdonsága nem szükségszerűen egyezik meg, félreértéseket kerülhetünk el, ha tervezési időben a Project Options párbeszédablak **Application** oldalán beállítjuk az alkalmazás nevét. Futásidőben ezt a következőképpen végezhetjük el:

```
Application.Title := Form1.Caption;
```

Az esetek többségében nem kell az alkalmazás-ablakkal foglalkoznunk azon túl, hogy beállítjuk címsorát és ikonját, viszont a 25. fejezetben több érdekes példát is láthatunk majd e globális objektum használatára.

Megjegyzés

A rejtett alkalmazás-ablak léte furcsa dolgokat képes előidézni, például a Delphiben írt alkalmazások főablaka nem rendeződik el rendesen, a 32 bites Windowsban azonban ez már nem olyan nagy gond.

Form-stílusok beállítása

A form tulajdonságai közül kettő igen nagy befolyással van az ablak viselkedésére: a FormStyle és a BorderStyle tulajdonságokról van szó. Az elsőn keresztül azt adhatjuk meg, hogy ablakunkat normál SDI ablakként, vagy MDI gyermekablakként akarjuk létrehozni.

A FormStyle tulajdonság lehetséges értékei:

- `fsNormal`: a form egy normál ablak, vagy párbeszédablak lesz.
- `fsMDIChild`: a form MDI gyermekablak.
- `fsMDIForm`: a form MDI szülőablakként jön létre, azaz az MDI alkalmazás keretablaka lesz (frame).
- `fsStayOnTop`: az ablak az összes többi ablakot el fogja takarni, kivéve a hasonló tulajdonsággal létrehozottakat.

Mivel az MDI szabványt követő alkalmazások alapvetően kétféle ablakstílust használnak (keret és gyermek), a FormStyle tulajdonság is két különböző értékkel jelzi ezeket. MDI alkalmazásokkal a 15. fejezetben foglalkozunk, most vizsgáljuk meg az utolsó, fsStayOnTop stílust!

Legfelül maradó form készítése

Egy mindig legfelül maradó (topmost) form készítéséhez elég a FormStyle tulajdonság megfelelő beállítása. A használt formtól függően ez a művelet két különböző eredmény hozhat:

- Ha az alkalmazás főformjára alkalmazzuk, akkor ez az ablak mindig a többi program ablaka fölött fog elhelyezkedni (hacsak nincsenek más alkalmazásban is ilyen „felső" formok).
- Egy másodlagos form esetén a form csak az adott alkalmazás ablakait fogja eltakarni.

Mielőtt egy példát megnéznénk, gondoljunk bele a felhasználási lehetőségekbe. Egy ilyen „felső" ablak bizonyos esetekben nagyon hasznos lehet, de néha igen zavaró is, hiszen ha teljes méretűre nyitjuk, akkor az alkalmazások váltása az ALT+TAB billentyűkkel tulajdonképpen láthatatlan. Ezért azt ajánlom, hogy csak másodlagos ablakoknál, eszköztáraknál használjuk e stílust. A Delphi egyes fejlesztőeszközeit is beállíthatjuk úgy, hogy mindig felül maradjanak.

Egy „felső" ablak készítése nagyon egyszerű. A Top példában egy menüponttal váltogathatjuk az ablak stílusát normálról „felsőre". Az alkalmazás **Help** menüjének **About** parancsával tesztelhetjük az egyes stílusok hatását.

Ha kiválasztjuk a **Style | Stay on Top** parancsot, az ablak „felső" stílusúvá válik:

```
procedure TForm1.StayOnTop1Click(Sender: TObject);
begin
  if FormStyle = fsStayOnTop then
    FormStyle := fsNormal
  else
    FormStyle := fsStayOnTop;
  StayOnTop1.Checked := not StayOnTop1.Checked;
end;
```

Mikor ez a metódus lefut, egy apró villanást észlelhetünk. Ez azért történik, mert a Delphinek meg kell semmisítenie és újra létre kell hoznia az ablakot, ha e tulajdonságot megváltoztatjuk.

A **Help** menü két eleme egy-egy üzenetet jelenít meg, de különböző technikákat használ ehhez. Az első a MessageDlg VCL függvényt hívja meg, a második a MessageBox API függvényt. Meglepő módon a második jól működik, az első azonban nem. A VCL dokumentációjához fordulva megtudhatjuk, hogy egy modális párbeszédablak „felső" formból való meghívása előtt *normalizálnunk* kell a formot. Ez – elvileg – az alábbi kóddal lehetséges:

```
procedure TForm1.AboutMessageDlg1Click(Sender: TObject);
begin
  Application.NormalizeTopMosts;
  MessageDlg ('This is a MessageDlg',
    mtInformation, [mbOK], 0);
  Application.RestoreTopMosts;
end;
```

Így azonban nem működik a dolog, legalábbis a Delphinek általam használt verziójában nem. Ezt a hibát valószínűleg egy későbbi javítóverzióban kiküszöböli a Borland.

Figyelem!

Mikor egy „felső" form modális üzenetablakot jelenít meg, az üzenet ablaka a form *mögé* kerülhet, viszont – mivel az üzenetablak modális – a formmal semmit nem tudunk csinálni, még mozgatni sem lehet! Egyetlen lehetőségünk, hogy az ALT+F4 billentyűkkel „vakon" lezárjuk az üzenetablakot.

A villanás kiküszöbölése

A zavaró villanás eltüntetéséhez egy pillanatra felejtsük el a Delphi által kínált tulajdonságokat, és a használjuk Windows API eszközeit. Több mód közül is választhatunk. A „felső" ablakstílus valójában az API ws_ex_Topmost konstansának felel meg, melyet – egyéb attribútumokkal egyetemben – futásidőben a SetWindowLong függvénnyel állíthatunk be.

Másik lehetőség a SetWindowPos függvény használata, mely ugyan hét paramétert vár, de esetünkben könnyen használható. Tulajdonképpen néhány jelzőbit (flag) beállításával lehetséges az ablak „felsővé" tétele:

```
SetWindowPos (Handle, hwnd_NoTopMost, 0, 0, 0, 0,
    swp_NoMove or swp_NoSize)
```

Ellentétes hatás elérése, azaz a „felső" stílus eltávolítása a hwnd_TopMost-ról hwnd_NoTopMost-ra cserélésével érhető el. Példánk új verziójának a Top2 nevet adtuk, és kívülről csak abban különbözi az előzőtől, hogy alapértelmezésben normál stílusú az ablak. Íme, a StayOnTop1Click metódus kódja:

```
procedure TForm1.StayOnTop1Click(Sender: TObject);
begin
  if StayOnTop1.Checked then
    SetWindowPos (Handle, hwnd_NoTopMost,
```

```
        0, 0, 0, 0, swp_NoMove or swp_NoSize)
    else
      SetWindowPos (Handle, hwnd_TopMost,
        0, 0, 0, 0, swp_NoMove or swp_NoSize);
    StayOnTop1.Checked := not StayOnTop1.Checked;
  end;
```

A fenti kódból kiderül, hogy a menüelem be- vagy kikapcsolt állapotából tudjuk meg a form stílusát, mivel az ablak FormStyle tulajdonsága mindig fsNormal marad, mégha a „felső" stílust használjuk is. Emlékezzünk csak vissza: bizonyos tulajdonságok a Windows értékeit tükrözik, vannak azonban olyanok is, melyek csak Delphiben léteznek, nincs Windows alatt megfelelőjük. Ez utóbbiakhoz tartozik a form stílusa is, és ezért nem változik a FormStyle tulajdonság értéke.

Ha összehasonlítjuk a két példát, észrevehetjük, hogy a második verzió nem villog. Bár a Delphi nagyszerű környezet, néhány nem kívánt „szolgáltatását" csak API függvények használatával kerülhetjük ki. A fenti technikák ismeretében csodálkozhatunk, hogy a Delphi miért nem ezeket a lépéseket követte. Nos, a SetWindowPos függvény igazából csak bizonyos körülmények közt használható, nem működik megfelelően másodlagos formokkal, ezenkívül ezzel sem lehet kivédeni a modális üzenetablakok és „felső" formok furcsa viselkedését.

Összefoglalásképpen kijelenthetjük, hogy ha csak egy ablakunk van, akkor használjuk a SetWindowPos függvényt, de legjobb, ha kerüljük elsődleges form „felső" ablakká tételét.

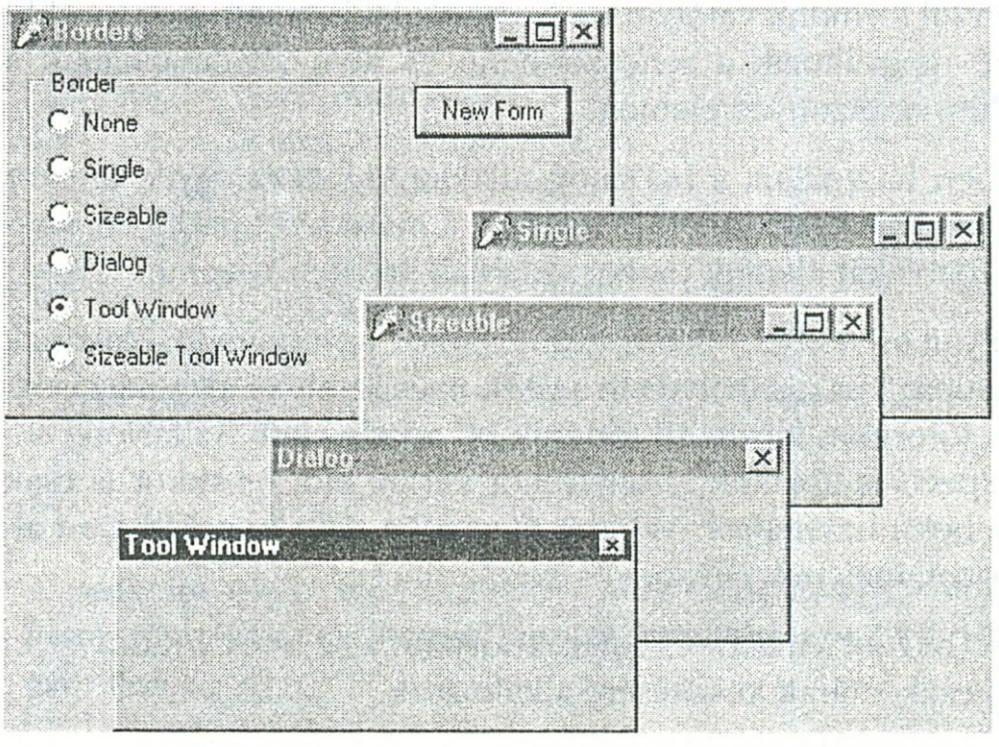
Megjegyzés

A SetWindowPos egy furcsa függvény. Szüksége van a kezelni kívánt ablak handle-jére, második paraméter ugyancsak egy ablak-handle *vagy* egy állapotjelző (ezzel állítjuk be az ablak *z-orderben* elfoglalt helyét, ez a képernyő síkjára merőleges z-tengelyt jelenti). Harmadik és negyedik paraméterében adható meg az ablak bal felső sarkának koordinátája, a következő két paraméter a szélességet és a magasságot jelenti, végül használhatjuk újabb jelzőbiteket logikai értékeit. Ami igazán különös, hogy az állapotjelzők értékétől függ, hogy egy adott paraméter használatba kerül-e vagy sem. Például, ha az utolsó paraméterként az swp_NoMove-ot is megadjuk, a harmadik és negyedik paraméter nem lesz figyelembe véve.

Keretstílus

A második tulajdonság, melyet a formokkal kapcsolatosan mélyebben kívánok ismertetni, a BorderStyle. Ez az ablakok keretét, mint látható elemet befolyásolja, de nagy hatással van az egész ablak viselkedésére is. A BorderStyle tulajdonság hat értéket vehet fel (10.2. ábra):

- `bsSizeable`: a form a normális, vastag keretet kapja, melynek segítségével az ablak méretezhető. Ez az alapértelmezett stílus.
- `bsDialog`: az ablaknak a fentihez hasonló vastag kerete lesz, de ez nem méretezhető – azaz egy párbeszédablakot kapunk.
- `bsSingle`: a form vékony keretet kap, mely nem méretezhető. A Windows 3.1-től eltérően Windows 95 alatt nincs észrevehető különbség a vékony és vastag keret között.
- `bsToolWindow`: az ablak vékony címsort kap, kisebb méretű felirattal és egy miniatűr lezárógombbal. Ezt a különleges stílust főleg nem méretezhető eszközpalettáknál használják.
- `bsSizeToolWin`: az előzővel megegyezik, de méretezhető.
- `bsNone`: a formnak sem kerete, sem egyéb szokványos építőeleme (címsor, rendszermenü) nem lesz.



10.2. ábra *Példák a különböző keretstílusokra*

A BorderStyle tulajdonság változtatás tervezési időben nem hoz eredményt, hiszen hogy is tudnánk átállítani egy ablak méretét, ha azt párbeszédablakká alakítottuk. Természetesen futás közben a form a kiválasztott keretet fogja tartalmazni.

Megjegyzés

Más komponenstulajdonságok is csak futási időben érvényesülnek, mivel hatásuk esetleg gátolna minket a tervezésben. Ilyen a Visible tulajdonság is: ha ez azonnal, fejlesztés közben érvényre jutna, a form eltűnne, és mi nem tudnánk tovább dolgozni vele.

A keretstílus hatása

Fejlesztés közben a formok mindig méretezhetőek. Ez a Windows vastag keretének felel meg. Ha egy ablaknak vastag kerete van, a felhasználó a keret segítségével méretezheti azt. A különleges méretező egérkurzor (kétirányú nyíl) jelenik meg, ha az egérmutatót az ablak kerete fölé visszük.

A másik nagyon fontos stílus a bsDialog. Ha ez választjuk, formunk a tipikus párbeszédablak-keretet fogja használni – vastag, de nem méretezhető. A grafikus különbségeken túl formunk valóban párbeszédablakká válik, s ez további változásokat jelent: megváltozik a rendszermenü és nem alkalmazhatjuk a BorderIcons tulajdonság bizonyos elemeit.

Általában két esetben használjuk a bsDialog tulajdonságot: az egyik, amikor „valódi" párbeszédablakot készítünk (13. fejezet), a másik eset pedig az, amikor az alkalmazás főablakát akarjuk párbeszédablak-szerűen kezelni.

A bsSingle stílust olyan esetekben alkalmazzák, amikor a program főablakának méretezése értelmetlen. Tipikusan ilyenek a játékok és az olyan alkalmazások, ahol közvetlenül a főformon keresztül történik pl. adatbevitel. Az fsNone stílust csak nagyon speciális ablakok készítésekor veszik elő, és akkor is csak normál ablakokon belül használják. Valószínűleg soha nem fogunk olyan alkalmazással találkozni, melynek címsora és kerete sincsen.

Az utolsó két stílust (bsToolWindow és bsSizeToolWin) a 32 bites Windows vezette be, és lebegő eszköztárak készítésére alkalmasak.

A Borders példa

A BorderStyle tulajdonság különböző értékeinek tesztelésére és megfigyelésére a Borders példaprogramot használhatjuk. Egyfajta kimenetét a 10.2. ábrán már láthattuk, de érdemes eljátszadozni vele, míg teljesen ki nem ismerjük az egyes beállítások hatását.

Az alkalmazás induló ablaka rádiógombokat és egy nyomógombot tartalmaz. Létezik egy másodlagos form is, Position tulajdonsága poDefaultPosOnly-ra állítva, ez a létrejövő form elhelyezkedését szabja meg (ezt a tulajdonságot később közelebbről is megvizsgáljuk).

```
object Form1: TForm1
  ...
  object BtnNewForm: TButton
    Left = 192
    T[illegible] = 16
    W[illegible] = 75
    [illegible] = 25
    [illegible] = 'New Form'
    [illegible]er = 0
    O[illegible]k = BtnNewFormClick
  end
  object BorderRadioGroup: TRadioGroup
    Left = 8
    Top = 8
    Width = 169
    Height = 153
    Caption = ' Border '
    ItemIndex = 2
    Items.Strings = (
      'None'
      'Single'
      'Sizeable'
      'Dialog'
      'Tool Window'
      'Sizeable Tool Window')
    TabOrder = 1
  end
end
```

A program kódja igen egyszerű: ahányszor megnyomjuk a gombot, egy új form jön létre, melynek kerettulajdonságai a rádiógombok beállításán alapulnak:

```
procedure TForm1.BtnNewFormClick(Sender: TObject);
var
  NewForm: TForm2;
begin
```

```
    NewForm := TForm2.Create (Application);
    NewForm.BorderStyle := TFormBorderStyle (
      BorderRadioGroup.ItemIndex);
    NewForm.Caption := BorderRadioGroup.Items[
      BorderRadioGroup.ItemIndex];
    NewForm.Show;
end;
```

A fenti kód egy apró trükköt alkalmaz: a kiválasztott elem sorszámát a TFormBorderStyle felsorolásba alakítja át. Ez azért ilyen egyszerű, mert a rádiógombok jelentésének és a felsorolt típus elemeinek sorrendje megegyezik:

```
type
TFormBorderStyle = (bsNone, bsSingle, bsSizeable,
    bsDialog, bsToolWindow, bsSizeToolWin);
```

Ezután BtnNewFormClick metódus a rádiógomb szövegét az form címsorába másolja. E program a SECOND.PAS néven elmentett TForm2 osztályát használja, ezért a főform egységének implementációjában el kell helyeznünk egy hivatkozást a második egységre:

```
uses
  Second;
```

Tipp

Mint már az 5. fejezetben említettem, inkább az egység implementációs részében helyezzünk el saját hivatkozásokat más egységekre, ne az interface részben. Ez felgyorsítja a fordítást és tisztább kódot eredményez, mivel a számunkra érdekes egység-hivatkozások elkülönülnek a Delphi által generáltaktól. Egy további előny, hogy így elkerülhetjük a körkörös hivatkozásokat (amikor két egység egymásra hivatkozik).

A keret ikonjai

Szintén fontos szerepet töltenek be a keret ikonjai. A Windows 3.1 ablakai alapértelmezésben rendszermenüt, minimalizáló és maximalizáló gombokat tartalmaztak. A Windows 95 a rendszermenü unalmas négyzetét ikonra cserélte, valamint megjelent a lezárógomb és a súgógomb, a címsor jobboldalán.

A BorderIcons tulajdonság igazából egy halmaz, melynek elemei a következők lehetnek: biSystemMenu, biMinimize, biMaximize és biHelp.

Megjegyzés

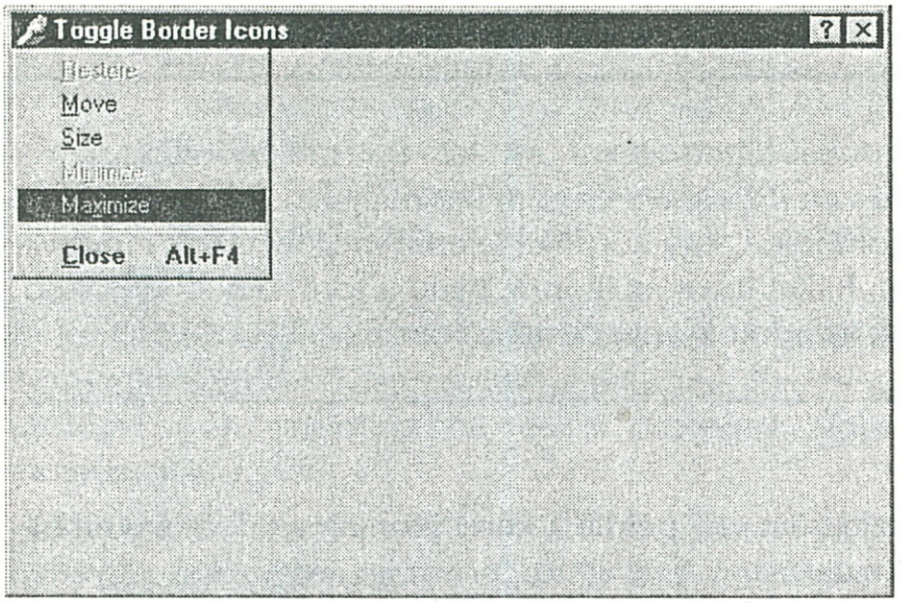
Az új biHelp ikonnal lehetséges a „Mi ez?" típusú súgó elkészítése. Ha ez a stílus be van kapcsolva és nem használjuk a minimalizáló és maximalizáló gombokat, egy kérdőjel jelenik meg az ablak címsorában, jobb oldalt. Rákattintva az ikonra, majd a form egy komponensére, megjelenik az adott komponenshez tartozó súgószöveg.

Természetesen ismét készítettem egy példát a keret ikonjainak demonstrálására, és azok futásidejű változtatására. A program formja igen egyszerű, egyetlen menüjében tudjuk kiválasztani a látni kívánt ikonokat.

A menüparancsok kiválasztásakor a megfelelő ikon ki- vagy bekapcsolódik. Írhattunk volna külön eseménykezelőket a négy menüponthoz, de én egyetlen eljárásba sűrítettem össze a teljes funkcionalitást. E kód igen jó példa a halmazok kezelésére is:

```
procedure TForm1.SetIcons(Sender: TObject);
var
  BorIco: TBorderIcons;
begin
  (Sender as TMenuItem).Checked :=
    not (Sender as TMenuItem).Checked;
  if SystemMenu1.Checked then
    BorIco := [biSystemMenu]
  else
    BorIco := [];
  if MaximizeBox1.Checked then
    Include (BorIco, biMaximize);
  if MinimizeBox1.Checked then
    Include (BorIco, biMinimize);
  if Help1.Checked then
    Include (BorIco, biHelp);
  BorderIcons := BorIco;
end;
```

A BIcons alkalmazást futtatva könnyen cserélgethetjük az ikonokat. Már első pillantásra látszik, hogy bizonyos elemek szoros kapcsolatban állnak egymással: ha eltávolítjuk a rendszermenüt, minden ikon eltűnik; ha a maximalizáló vagy minimalizáló gombok egyikét nem használjuk, az szürke ikonként jelenik meg; ha egyiket sem használjuk, teljesen eltűnnek. Utóbbi két esetben a rendszermenü megfelelő elemei is változnak. A minimalizáló és maximalizáló gombok eltávolítása után tudjuk megjeleníteni a Súgó ikont (10.3. ábra).



10.3. ábra *A BIcons példában látható Súgó ikon. Érdemes megfigyelni, hogy a rendszermenü elemei is az ikonok állapotát tükrözik*

További ablakstílusok használata

A keret stílusát és ikonjait két tulajdonságon keresztül változtathatjuk. A szemmel látható változásokon túl ezek az ablakok működését is befolyásolják. Fontos tudnunk, hogy e beállítások különböző értékei az ablakok stílusát és kiterjesztett (extended) stílusát jelentik. A két megnevezés a CreateWindowsEx API függvény két paraméterének felel meg, melyet a Delphi az ablakok létrehozására használ. Jó, ha tudjuk, hogy a CreateParams virtuális metódus felüldefiniálásával megváltoztathatjuk e két paramétert:

```
public
  procedure CreateParams(
    var Params: TCreateParams); override;
```

Ez az egyetlen módja olyan ablakstílusok használatának, melyek a megefelelő tulajdonságokon keresztül nem állíthatók be. A súgóban a CreateWindow és CreateWindowEx függvények ismertetésénél találhatjuk meg a használható ablakstílusok listáját. A Win32 API számos új ablakstílust támogat, például az eszközpaletták ablakait.

Készítettem egy kis példát Transpar néven, melyben egy tökéletesen használhatatlan átlátszó ablakot készítek:

```
procedure TForm1.CreateParams(var Params:TCreateParams);
begin
  inherited CreateParams (Params);
  Params.ExStyle:=Params.ExStyle or ws_ex_Transparent;
end;
```

A program futtatása közben elgondolkodhatunk, valyon mire jó egy olyan ablak, mely még a keretét sem rajzolja ki. Nos, kicsit én is zavarban vagyok, bár el tudok képzelni olyan esetet, amikor egy nagy átlátszó ablakkal letakarunk egy másikat, hogy elfogjuk az összes egéreseményt.

Egyéni ablakstílusok beállításánál sokkal gyakoribb ok egy program induló képernyőjének (splash screen) elkészítése. Ilyenkor eltávolítjuk az ablak címsorát, mint ahogy a 13. fejezetben látni fogjuk. Mindezek ellenére speciális ablakstílusokat sokkal többször használunk új komponensek létrehozásánál, mint a normál ablakoknál.

Ha egy formon sok komponenst helyezünk el, általában nem méretezhető tesszük, nehogy méretezés után egy komponens kívül kerüljön az ablak látható területén. Ám az sem nagy baj, ha ez történt, mivel a Delphi ilyenkor automatikusan görgetősávokat ad az ablakhoz (a görgetés lesz a 14. fejezet egyik fő témája).

Nagy formok készítésekor figyeljünk a következőkre: attól, hogy mi nagy felbontásban dolgozunk, még nem következik, hogy a felhasználó is így használja számítógépét. Bár ez apróságnak tűnhet, többszer előfordul, mint gondolnánk. Ha lehet, ne készítsünk 640 × 480 pixelnél nagyobb formokat (elsősorban a párbeszédablakokra kell figyelni).

Ha mégis nagy ablakokat kell készítenünk, és a görgetősávok sem jelentenek megoldást, kihasználhatjuk a Delphi két ügyes lehetőségét:

- A form ScaleBy tulajdonságával az ablakot és komponenseit egyszerre méretezhetjük. Ennek értékét általában a program indulásakor állítjuk be, miután megállapítottuk a képernyő felbontását.
- A PixelsPerInch és Scaled tulajdonságok értékei alapján a Delphi automatikusan méretezi a formokat, az adott felbontástól függően. Természetesen e két tulajdonság értékét is változtathatjuk futásidőben, ahogyan később látni fogjuk.

Mindkét esetben ügyeljünk arra, hogy a form AutoScroll tulajdonsága False-ra legyen állítva. Ha nem így teszünk, csak a form tartalma méreteződik, kerete nem.

A formok manuális méretezése

A formok méretezéséhez mindig használhatjuk a ScaleBy metódust, amelynek két egész paramétere van: egy számláló és egy nevező, vagyis egy tört. Ezt a módszert az egyszerű komponenseknél is alkalmazhatjuk. Például ha a következő programsort írjuk be:

```
ScaleBy (3, 4);
```

akkor az aktuális form méretét osztjuk néggyel és szorozzuk hárommal, vagyis az eredeti méret háromnegyedére csökkentjük. Általában célszerűbb százalékértékeket használni, valahogy így:

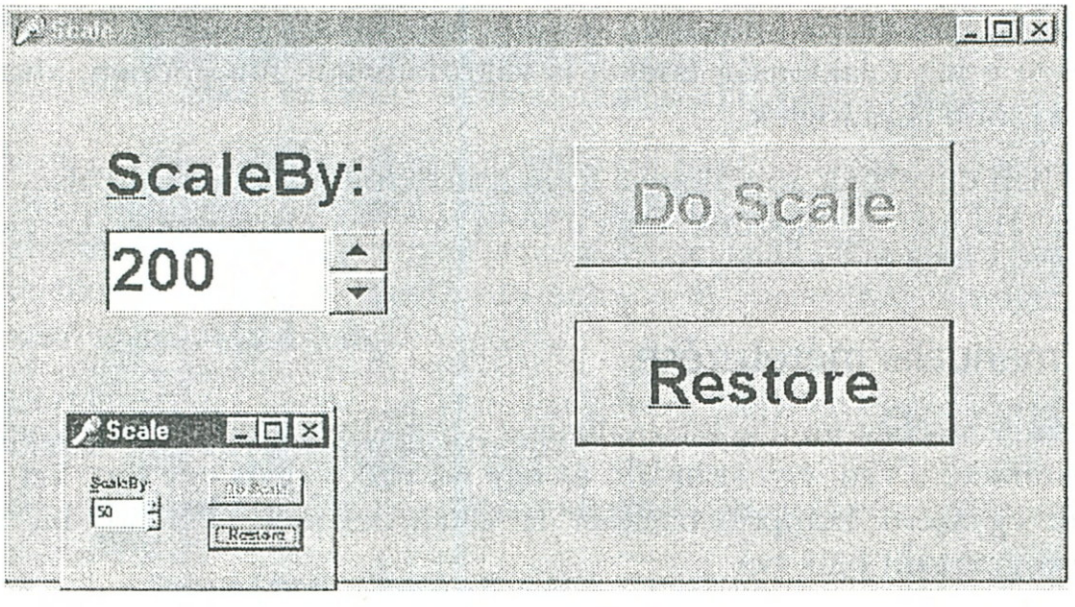
```
ScaleBy (75, 100);
```

A form átméretezése aránytartó, de egy bizonyos határ alatt a feliratok arányai kissé torzulhatnak. Azt se feledjük, hogy ha túlságosan lekicsinyítjük a formot, akkor a komponensek többsége használhatatlan lesz, esetleg el is tűnik. Ezt a problémát az okozza, hogy a Windows komponenseinek helyzetét és méretét csak egész számokkal lehet megadni, ez a fajta méretezés pedig szinte mindig tört számokat eredményez, amiket persze kerekíteni kell.

Az ilyen típusú problémákat úgy védhetjük ki, hogy határt szabunk a felhasználó méretezési lehetőségeinek, vagy egy vázlat alapján minden méretezés előtt újra elkészítjük a formot, hogy a kerekítési hibák ne gyűljenek fel.

Figyelem!

Ha a ScaleBy metódust alkalmazzuk a formra, akkor valójában nem a form, hanem csupán a komponensei változtatják méretüket. Ahogy már korábban is említettem, ezt a form AutoScroll tulajdonságának kikapcsolásával lehet megkerülni. De mi az összefüggés a méretezés és a görgetés között? Nos, véleményem szerint az, hogy ha a görgetés engedélyezve van, akkor a komponenseket a form látható területén kívülre is mozgathatjuk, máskülönben maga a form is nőni fog.



10.4. ábra *A Scale példa formja, ha 50-nel és 200-zal méreteztük*

Egy egyszerű példán keresztül szeretném bemutatni, hogyan lehet manuálisan méretezni a formot úgy, ahogy azt a felhasználó kívánja. A példa címe Scale lesz. A program formján (lásd 10.4. ábra) két gomb, egy címke, egy szerkesztőmező, és egy eddig még nem használt komponens, az UpDown (léptetőgomb) vezérlő található.

Az UpDown komponenst az Associate (társít) tulajdonsággal a szerkesztőmezőhöz kapcsoljuk. Mostantól a felhasználó csak számokat írhat a mezőbe, vagy a mező melletti fel és le gombokkal egy fix mennyiséggel növelheti vagy csökkentheti annak tartalmát (ezt a mennyiséget az UpDown komponens Increment (növekmény) tulajdonságánál állíthatjuk be). A beállított értéket a szerkesztőmező Text (szöveg), vagy a léptetőgombok Position (helyzet) tulajdonságával érhetjük el. A beállítatható tartományt is megadhatjuk, ahogy azt a form leírásából is láthatjuk:

```
object Form1: TForm1
  AutoScroll = False
  Caption = 'Scale'
  Font.Height = -16
  Font.Name = 'Arial'
  Font.Style = [fsBold]
  object Label1: TLabel
    Caption = '&ScaleBy:'
  end
  object ScaleButton: TButton...
  object RestoreButton: TButton...
  object UpDown1: TUpDown
    Associate = Edit1
    Min = 30
    Max = 300
    Increment = 10
    Position = 100
    Wrap = False
  end
end
```

A ScaleButton gomb megnyomásakor az aktuális beállított értéket használjuk a méretezés arányának meghatározására:

```
procedure TForm1.ScaleButtonClick(Sender: TObject);
begin
  AmountScaled := UpDown1.Position;
  ScaleBy (AmountScaled, 100);
  UpDown1.Height := Edit1.Height;
  ScaleButton.Enabled := False;
  RestoreButton.Enabled := True;
end;
```

Ez az eljárás a beállított értéket a form AmountScaled nevű privát mezőjében tárolja, bekapcsolja a Restore (visszaállítás) gombot, és kikapcsolja a megnyomottat. Később, ha a felhasználó megnyomja a Restore gombot, az elvégzett méretezés ellenkezőjét hajtja végre a program a ScaleBy (100, AmountScaled) eljárás meghívásával. Mindkét esetben egy-egy külön sor gondoskodik arról, hogy az UpDown komponens Height (magasság) tulajdonsága megegyezzen a hozzá tartozó szerkesztőmezőével.

Tipp

Ha azt szeretnénk, hogy a formok szövegei, a komponensek feliratával és a listapanelek elemeivel együtt helyesen változtassák méretüket, akkor kizárólag TrueType fontokat használjunk. A rendszerfontok és a bittérképes betűkészletek nem méreteződnek helyesen. Ez a probléma fontos, mert sok komponens mérete függ attól, hogy mekkora a feliratuk, és ha a felirat nem megfelelő méretű, a vezérlő mérete is rosszul változhat. Ilyen meggondolásból én például az Arial betűtípust használtam.

Formok automatikus méretezése

Arra is lehetőségünk van, hogy az előbbi módszer helyett a Delphi-re bízzuk a munkát. Induláskor a Delphi lekérdezi a rendszertől a használt képernyőfelbontást, és eltárolja azt a Screen (képernyő) objektum PixelsPerInch tulajdonságában. A Screen objektum a VCL egy különleges eleme, amely minden alkalmazás számára elérhető. Tervezési időben a PixelsPerInch értékét a használt form is tartalmazza, csak olvasható értékként. Ha futtatjuk az alkalmazást, és a form Scaled tulajdonsága igazra van állítva, akkor a Delphi automatikusan a megfelelő méretűre alakítja a formot.

Így az alkalmazás automatikusan a használt felbontáshoz igazítja saját méretét anélkül, hogy egy sorral is bővítenénk programunkat. Ez az automatikus méretezés ugyan néha kisebb problémákhoz vezethet, ám ezek elkerülhetők, ha betartjuk a következő szabályokat:

- Állítsuk a formok Scaled tulajdonságát igazra.
- Csak TrueType fontokat használjunk.
- A Windows kis fontjait használjuk fejlesztésnél.
- Állítsuk az AutoScroll tulajdonságot hamisra, ha magát a formot is, és nem csupán annak vezérlőit akarjuk méretezni.

Méretezés és fontok

Mind az automatikus, mind pedig a ScaleBy metódussal végzett méretezés a komponensek fontjának nagyságának megváltoztatásával történik. Így tehát a vezérlők mérete az általa használt betűtípustól függ.

Automatikus méretezéskor a form PixelsPerInch tulajdonsága (a tervezési időben felvett érték) összehasonlításra kerül az aktuális rendszerértékkel (amit a Screen objektum megfelelő mezője mutat), és az eredményt használja a Delphi a komponensek fontjának beállítására. Pontosan ez úgy történik, hogy a szöveg tervezési időbeni és végleges magassága kerül összehasonlításra, és ha nem egyeznek meg, akkor történik méretigazítás.

Nem szabad elfelednünk, hogy a méretezés elsősorban a fontméreten, nem a képernyőfelbontáson alapszik. Még ha két számítógép felbontása meg is egyezik (például 800×600), lehet hogy az egyiken a PixelPerInch arány 96, mert ott kis fontokat használnak, a másikon pedig 120, mert ott nagyok a betűk. Ez azt jelenti, hogy a program alkalmazkodni próbál majd a megváltozott körülményekhez.

A form helyzetének és méretének állítása

Ha nem az előbbiekben ismertetett módszerrel dolgozunk, akkor nyilván elvárjuk, hogy formunk úgy jelenjen meg a képernyőn, ahogy tervezéskor. Alapértelmezésben ez is fog történni, de van arra lehetőségünk, hogy ezen változtassunk.

Az itt használható egyik tulajdonság a Position (helyzet) nevet viseli, ez a form kezdeti pozícióját jelöli a képernyőn. A megadható értékek pontos jelentése a Windows aktuális környezeti beállításaitól is függ, például attól, hogy milyen elrendezésben jelennek meg a képernyőn az új ablakok. Íme a tulajdonság lehetséges értékei:

- `poDesigned`: A form úgy jelenik meg, ahogy azt mi tervezési időben megadtuk neki. Az ide vonatkozó tulajdonságok a Left (bal), Top (teteje), Height (magasság) és a Width (szélesség), amelyeket megadhatunk számértékkel is, de általában a form méretezésével egyszerűbb beállítani a kívánt méretet.
- `poDefault`: a Windows határozza meg a form helyzetét és méretét úgy, hogy az igazodjon a lépcsőzetes elrendezéshez, függetlenül attól, hogy mit állítottunk be tervezési időben. Ha egymás után többször futtatjuk alkalmazásunkat, akkor az új ablakok a képernyő jobb alsó sarka

felé közeledve jelennek meg, és ha méretezhetőek, akkor nagyságuk is megváltozik. Ha elég sok ablakot nyitunk meg, akkor a folyamat visszatér a kezdőpozícióba, és újra indul. Fontos megjegyezni, hogy ez a lehetőség csak főablakoknál (bsSizeable vagy bsFixed) működik, párbeszédablakoknál nem.

- `poDefaultPosOnly`: A form a tervezési időben megadott méretben, de a Windows által, az előbb ismertetett algoritmus alapján meghatározott helyen jelenik meg. Magam is ezt a módszert használtam a Borders példa másodlagos formjainak elkészítésekor.
- `poDefaultSizeOnly`: A form a tervezési időben beállított helyen, de a Windows által meghatározott méretben jelenik meg. A form jobb széle a képernyő jobb oldalához, az alsó széle pedig a képernyő aljához van mindig közel, a helyzettől függetlenül. Ezt az értéket ritkán használjuk.
- `poScreenCenter`: A form a tervezési időben beállított méretet használva mindig a képernyő közepén jelenik meg.

Figyelem!

Ha Delphi alkalmazásainkat nagyfelbontású módban készítjük, ne hagyjuk a formok Position tulajdonságát poDefault értéken, különben a formok jobb alsó sarka nem fog látszódni alacsonyabb felbontású képernyőkön.

A második paraméter, ami hatással lehet az ablak kezdeti méretére és helyzetére az ablak állapota. Tervezéskor a WindowState tulajdonság segítségével adhatjuk meg, hogy induláskor teljes méretű, vagy kis méretű legyen a form. Ennek a tulajdonságnak összesen három értéke lehet: wsNormal (normális méret), wsMinimized (kis méret) és wsMaximized (teljes méret). Ha kis méretűnek állítjuk be formunkat, akkor az a Windows 95 Tálcáján fog csak megjelenni, ha teljes méretűvé, akkor az egész képernyőt elfoglalja.

Megjegyzés

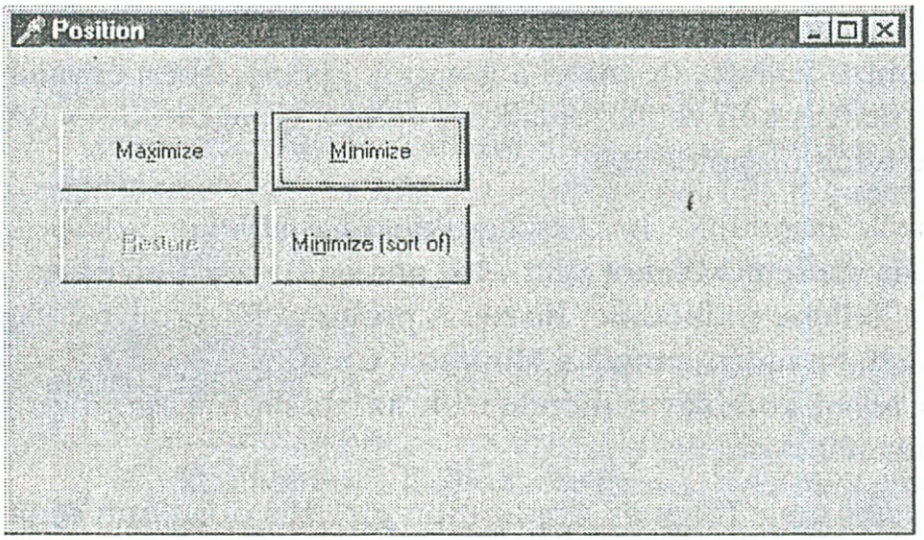
Vegyük észre, hogy ha átállítjuk a címsor ikonjainak alapértékeit vagy fix méretűre állítjuk a formot, akkor az mindig a WindowState tulajdonság által meghatározott módon jelenik meg. Más szóval, ha ezt a tulajdonságot wsMaximized-re állítjuk, akkor a dialógusablak teljes méretben jelenik meg, noha a párbeszédablakokat általában nem lehet méretezni.

A form kis- és teljes méretűvé alakítása

Természetesen futásidőben is változtathatjuk az ablak méretét, csupán a WindowState tulajdonságot kell átállítanunk. Pontosan ez történik az általam írt Position című példában is. A példa a rendszer által meghatározott méreteket és helyzeteket, valamint a WindowState tulajdonság futásidejű állítását mutatja be. A program egy egyszerű formból és négy gombból áll, ahogy azt a 10.5. ábrán is láthatjuk.

A következő eljárás fut le, ha a felhasználó megnyom egy gombot:

```
procedure TForm1.BtnMaximizeClick(Sender: TObject);
begin
  WindowState := wsMaximized;
end;
```



10.5. ábra *A Position példa formja. Figyeljük meg a gombok állapotát!*

A Restore gomb állítja a tulajdonságot wsNormal-ra. A két gomb kizárja egymást, így a nem elérhetőt mindig ki kell kapcsolnunk. Például ha az ablak teljes méretű, a Maximize gombnak nem lesz semmilyen hatása, úgyhogy ki is kapcsolhatjuk. Egyszerűnek tűnik, ugye? Sajnos a helyzet azért ennél bonyolultabb. Nem vettük ugyanis figyelembe, hogy az ablak méretét a címsorában található gombokkal is lehet állítani, így programunk nem fog helyesen működni. Közelítsük meg hát más oldalról a kérdést!

Eseményvezérelt rendszert használunk, keressünk hát egy eseményt, amellyel megoldhatjuk a problémát! Ebben a példában az OnResize esemény éppen megfelelő lesz: ezt hívja meg a Delphi minden olyan alkalommal, amikor a form mérete megváltozik. Íme a kódrészlet, amit ehhez az eseményhez társítottam:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  if WindowState = wsMaximized then
  begin
    BtnMaximize.Enabled := False;
    BtnRestore.Enabled := True;
  end
  else
  begin
    BtnMaximize.Enabled := True;
    BtnRestore.Enabled := False;
  end;
end;
```

Ne feledjük, hogy a fenti programrészlet lefut az alkalmazás indulásakor is, és ezért tervezési időben az összes gomb be van kapcsolva, a program gondoskodik induláskor a Restore gomb kikapcsolásáról. Már csak egy gombbal nem törődtünk, és ez a Minimize feliratú, de mivel a gombok kis méretben egyáltalán nem látszanak, így nem is kell velük foglalkoznunk. A gombok ki- és bekapcsolásának kérdése tehát megoldódott.

Van azonban egy másik gondunk: ha összecsukjuk az ablakot, akkor a Windows egy minimális méretű ablakot állít elő, amelyet nem helyez el a Tálcán. Ennek nem így kellene működnie, hiszen a program fő formjáról van szó. Ki is próbálhatjuk ezt, ha megnyomjuk a Minimize gombot. Abban az esetben viszont, amikor a fejléc gombjával méretezzük az ablakot, a program a Tálcára kerül, úgy ahogy azt elvárjuk.

A problémát úgy oldhatjuk meg, hogy az Application globális objektum segítségével állítjuk kis méretűre a formot:

```
procedure TForm1.BtnMinimizeClick(Sender: TObject);
begin
  Application.Minimize;
end;
```

Létezik egy Restore (előző méret) metódusa is a TApplication osztálynak, amit a form méretének visszaállítására használhatunk; bár az esetek döntő többségében a felhasználó ezt a megszokott Windows műveletként fogja végrehajtani (a rendszermenü Restore parancsával, vagy a Tálcán való kattintással).

Tipp

A kis és teljes méretre hozás általában együtt jár bizonyos grafikus és hangeffektusokkal. A Delphi alkalmazásokban ezeknek az effektusoknak a nagy része hiányzik, de könnyen megoldhatjuk ezt a problémát a PlaySound API függvény meghívásával.

A form és kliensterületének mérete

Tervezési időben a form méretét a széleinek húzásával, vagy Width és Height tulajdonságainak állításával állíthatjuk. Futási időben a felhasználó átméretezheti az ablakot az egérrel, de a program is megteheti ugyanezt az előbb említett tulajdonságok állításával.

Ha megnézzük a form tulajdonságait, észrevehetjük, hogy két különböző tulajdonság is vonatkozik annak szélességére, és kettő a magasságára:

- A Height a form magasságát adja meg, a kerettel együtt.
- A Width a form szélességét adja meg, a kerettel együtt.
- A ClientHeight a form belső területének magasságát adja meg, amely nem tartalmazza a keretet, a feliratot és a menüsort.
- A ClientWidth a form belső területének szélessége, a keret nélkül.

A fenti felsorolásból kettő a form kliens területére vonatkozik, amely az ablak belső, keret és menüsor nélküli része. Ez az a terület, amelyen elhelyezhetjük komponenseinket, ez fogadja a felhasználó által generált eseményeket, és itt jelenik meg a program kimenete.

Megjegyzés

A Windows-ban arra is van lehetőség, hogy a kliens területen kívülről, a keretről kapjunk bemeneti adatokat, és küldjünk kimenőket, ám ez egy igen összetett feladat. Ha érdekel bennünket a téma, akkor utánanézhetünk a Súgóban a dolognak. Az ide tartozó Windows üzenetek a wm_NCPaint, a wm_NCCalcSize, a wm_NCHitTest, és egy sor olyan egérműveletekhez kapcsolódó üzenet, mint a wm_NCLButtonDown. A dolog azért bonyolult, mert a Windows működését össze kell hangolnunk saját programunkkal, de szerencsére ezt a Delphi lehetővé teszi azzal, hogy ilyen alacsony szintű Windows üzeneteket is képes kezelni, és erre nem sok fejlesztőkörnyezet képes.

Néha szükségünk lehet a felhasználható terület pontos méretére, ilyenkor van értelme a kliens terület állításának. Ha a form kliens méretét változtatjuk, akkor természetesen vele együtt változik az egész form mérete is – ha például megnöveljük a ClientHeight értékét, akkor a Height értéke is azonnal megnő.

A formok maximális és minimális mérete

Ha formunkat méretezhetővé tesszük, akkor a felhasználók kedvük szerint méretezhetik azt, akár akkorára is, hogy a teljes képernyőt lefedje az ablak. A Windows a form méretének megváltozásáról a wm_Size üzenettel értesít bennünket, ami az OnResize eseményt generálja. Ez az esemény a form átméretezése után következik be. Nem igazán jó megoldás azonban a formot itt átméretezni, ha a felhasználó túl nagyra vagy túl kicsire változtatta az ablakot, inkább meg kellene előznünk valahogy ezt a dolgot.

Mielőtt megmutatom, hogyan kell egy ablak maximális és minimális méretét megadni, nézzünk meg néhány áthidaló ötletet arra, hogyan lehetne a fenti problémát megelőzni! Először is készíthetünk olyan ablakokat, amelyeket nem lehet átméretezni. Másodszor, ha a vezérlők túllógnak az ablak keretén, és a form AutoScroll tulajdonságát igazra állítjuk, akkor a Delphi automatikusan görgetősávokkal látja el azt, és így a komponensek elérhetővé válnak.

Persze sokszor hasznos lehet határt szabni a méretezésnek, különösen a kicsinyítésnek. A Delphi nem tartalmaz külön tulajdonságokat erre a célra, de a Windows üzeneteinek segítségével könnyen kezelhetjük ezt a problémát.

A nekünk kellő üzenet a wm_GetMinMaxInfo lesz. A megfelelő eljárás paraméterének típusa TWMGetMinMaxInfo kell hogy legyen (az üzenet paramétereinek típusai a Messages egységben vannak megadva). Ez a struktúra egy MinMaxInfo szerkezetre mutató pointert tartalmaz, amely a Windows egységben a következőképpen van definiálva:

```
type
  PMinMaxInfo = ^TMinMaxInfo;
  TMinMaxInfo = record
    ptReserved: TPoint;
    ptMaxSize: TPoint;
    ptMaxPosition: TPoint;
    ptMinTrackSize: TPoint;
    ptMaxTrackSize: TPoint;
  end;
```

A struktúra mezői igen összetettek, de nagyon jól használhatóak is:

- `ptReserved`: ez egy nem dokumentált mező, amit a Windows belső használatra tart fönn.
- `ptMaxSize`: ennek x mezőjében tároljuk azt szélességet, y mezőjében pedig azt a magasságot, amelyet teljes méretre hozáskor felvesz az ablak.
- `ptMaxPosition`: ez a pont jelzi azt a koordinátát, amely az ablak bal felső pontja lesz, ha azt teljes méretűre alakítjuk.

- `ptMinTrackSize`: ez jelzi az ablak minimális szélességét és magasságát, amelyet a felhasználó beállíthat.
- `ptMaxTrackSize`: az ablak maximális szélessége és magassága.

A struktúra mezőinek mindig van alapértelmezés szerinti értékük, így meg kell változtatnunk a számunkra fontosakat, ha megkapjuk az üzenetet. A MinMax példában megadtam a minimum és maximum értékeket, valamint kikapcsoltam a Maximize gombot. Annak, hogy lehetővé tesszük az ablak teljes méretre hozását anélkül, hogy az a teljes képernyőt eltakarná (10.6. ábra), csak ritkán van értelme (ilyen például a Delphi fő ablaka).

10.6. ábra *A MinMax példa ablaka igencsak apró, mégis rendelkezik a teljes méretű ablakok Előző méret gombjával*

A MinMax példa formja igen egyszerű, egyetlen komponens sincs rajta. Csupán annyit kell rajta változtatnunk, hogy a BorderIcons tulajdonság biMaximize pontját hamisra állítjuk. Az ablak méretezésének lekövetéséhez néhány sort be kell írnunk. A TForm1 osztály definíciójában meg kell adnunk a következő eljárást:

```
public
  procedure GetMinMax (var MinMaxMessage:
      TWMGetMinMaxInfo); message wm_GetMinMaxInfo;
```

Felhívom a figyelmet a message direktívára, amely az eljárást közvetlenül a Windows üzenethez kapcsolja. Második lépésként elkészítjük az eljárás kódját, amelyben a MinMaxMessage paraméter MinMaxInfo mezője által mutatott struktúra megfelelő értékeit állítjuk be. A `with` parancs használatával megkönnyíthetjük a különböző mezők elérését, értékük megváltoztatását:

```
procedure TForm1.GetMinMax (
  var MinMaxMessage: TWMGetMinMaxInfo);
begin
  with MinMaxMessage.MinMaxInfo^ do
  begin
    ptMinTrackSize.x := 150;
    ptMinTrackSize.y := 150;
    ptMaxTrackSize.x := 300;
    ptMaxTrackSize.y := 300;
  end;
end;
```

A program igen egyszerűen működik: a felhasználó nem méretezheti az ablakot csak bizonyos határok között, ahogy azt láthatjuk, ha futtatjuk a programot.

Formok készítése automatikusan

Egészen mostanáig nem esett szó a formok készítéséről. Azt már tudjuk, hogy a form létrehozásakor az OnCreate üzenetet kapjuk, amellyel megváltoztathatjuk vagy lekérdezhetjük a form kezdeti értékeit. Azt azonban eddig nem tudtuk befolyásolni, hogy a form létrejöjjön a program indulásakor, vagy sem. A létrehozásért felelős utasítás a projekt forrásállományában (vagy a **View | Project** menüvel elérhető .DPR fájlban) található:

```
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Ha át akarjuk lépni az automatikus létrehozást, akkor átírhatjuk ezt a részt, de még jobb, ha módosítjuk a Project Options párbeszédablak **Forms** oldalát (lásd 10.7. ábra). Itt adhatjuk meg azt, hogy a form automatikusan létrejöjjön-e, vagy sem. Ha ezt kikapcsoljuk, akkor a projekt inicializációs része a következőképpen módosul:

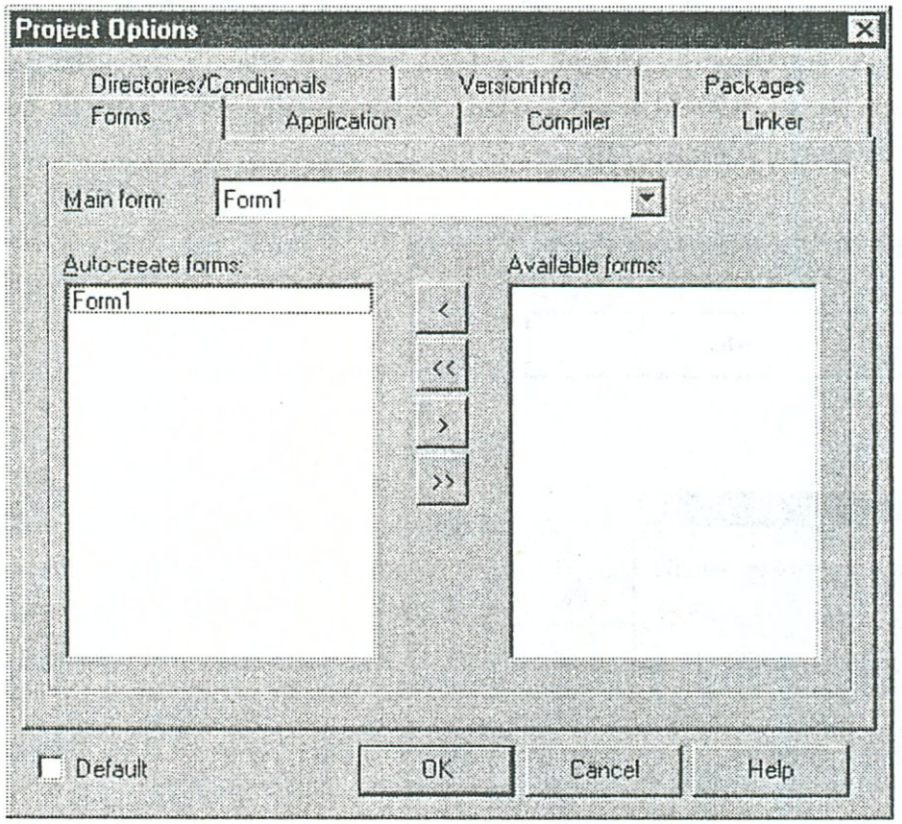
```
begin
  Applications.Initialize;
  Application.Run;
end.
```

Ha most futtatjuk a programot, akkor semmi nem történik, az alkalmazás azonnal megáll, hiszen nem készítettük el a főablakot. Hogyan is működik tehát az alkalmazás CreateForm metódusa? Úgy, hogy a neki átadott osztályt példányosítja, és a kapott objektumot értékül adja második paraméterének.

Valami más is történik azonban a színfalak mögött. Ha a CreateForm metódust úgy hívja meg a rendszer, hogy még nem létezik főformunk, akkor az aktuális

form értékül adódik az alkalmazás MainForm (főform) tulajdonságának. Ezért van az, hogy a 10.7. ábra párbeszédablaka az elsőként elkészített formot jelöli főformként a megfelelő helyen.

Az előbbiekben elmondottak igazak az alkalmazás bezárására is. A főform bezárásakor a program befejezi futását, a többi formtól függetlenül. Ha ezt a műveletet programból szeretnénk végrehajtani, akkor egyszerűen hívjuk meg a főform Close metódusát, ahogy azt korábbi példáinknál is tettük.



10.7. ábra *A Delphi Project Options párbeszédablakának Forms oldala*

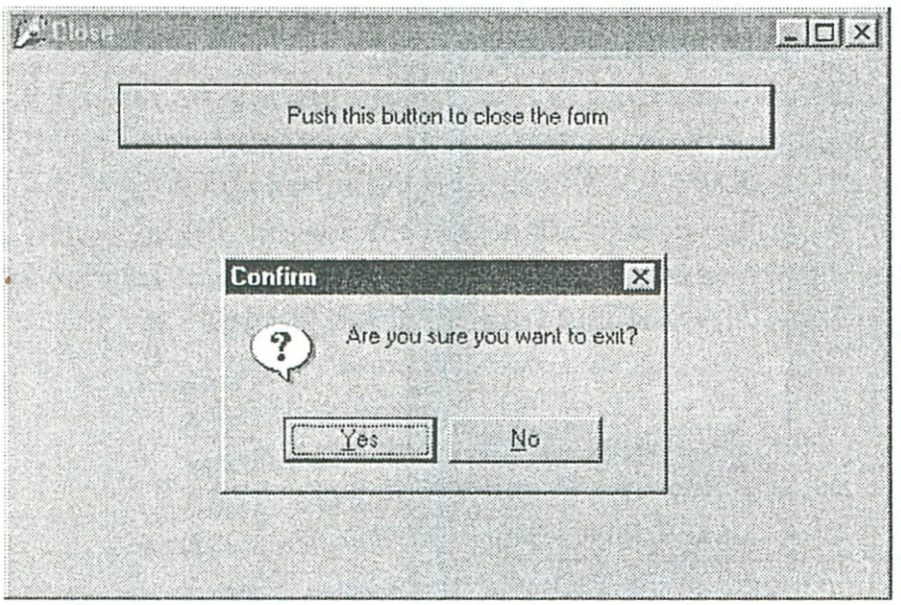
A form bezárása

Ha bezárunk egy formot az imént említett módszerrel vagy a szokásos módon (ALT+F4, a rendszer menü, vagy a Close gomb), az OnCloseQuery eseményt váltja ki a rendszer. Itt megkérdezhetjük a felhasználót, hogy biztosan be akarja-e zárni az alkalmazást, ez különösen akkor fontos, ha az elmentetlen adatokat tartalmaz. Az előző fejezet RichNot2 példája is ezt a módszert alkalmazza.

Csak emlékeztetőül, egy másik rövid példa, amely a Close című programban található:

```
procedure TForm1.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if MessageDlg ('Are you sure you want to exit?',
      mtConfirmation, [mbYes, mbNo], 0) = idNo then
    CanClose := False;
end;
```

A Close példa egy formból és egy rajta található gombból áll (az előbbi metódus meghívásának eredményét a 10.8. ábra mutatja). Ha az OnCloseQuery eredménye az, hogy a formot be kell zárni, akkor a következő meghívott esemény az OnClose lesz. A harmadik lépést az OnDestroy esemény jelenti, amely az OnCreate ellentéte, és általában a formhoz kapcsolt objektumok és memóriaterület felszabadítására használjuk.



10.8. ábra *A Close példa első párbeszédablaka*

Mi hát a közbülső OnClose esemény jelentőssége? Ebben a metódusban még megakadályozhatjuk az alkalmazás bezárását, esetleg különböző „bezárási műveleteket" adhatunk meg. A metódusnak van egy átadott Action nevű paramétere, amelynek értékével befolyásolhatjuk a form viselkedését. Ez a következő értékeket veheti fel:

- `caNone`: a form nem záródhat be. Ez a művelet az OnCloseQuery CanClose paraméterének hamisra állítására vonatkozik.
- `caHide`: a form nem záródik be, csupán el lesz rejtve. Ennek akkor van értelme, ha az alkalmazásnak más formjai is vannak; máskülönben a program futása véget ér. Ez a másodlagos formok alapértelmezés szerinti viselkedése.

- `caFree`: a form bezáródik, és a hozzá tartozó memóriaterület felszabadul. Ha ez a főform volt, akkor a program is befejeződik. Ez a főformok alapértelmezés szerinti viselkedése.
- `caMinimize`: a form kis méretű lesz, és nem záródik be. Ez az MDI gyermekablakok alapértelmezés szerinti viselkedése.

A Close példában írtam egy eljárást az OnClose eseményhez is. Annyi történik itt, hogy a program újra rákérdez, hogy valóban ki akarunk-e lépni belőle:

```
procedure TForm1.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  if MessageDlg (
    'Are you REALLY SURE you want to exit?' + Chr(13)+
    '(This is your last chance to remain with us!)',
    mtConfirmation, [mbYes, mbNo], 0) = idNo then
        Action := caNone
  else
    Action := caFree;
end;
```

Csupán azért készítettem el ezt a második programrészt, hogy megmutassam, mi a két lehetséges megoldása a kilépés megerősítésének. Nem ajánlom azonban hogy programjainkban ehhez hasonlót csináljunk, mert ezzel könnyen felidegesíthetjük a felhasználót. Annak lehet esetleg értelme, ha az OnClose eseményből megkérdezzük, hogy a program többi formja zárva van-e már (az OnCloseQuery eseménykezelő meghívásával). Általában azonban a bezárásra vonatkozó kérelem elbírálását célszerű az OnCloseQuery eseménynél elvégezni, a bezáráskor felmerülő különleges műveleteket pedig az OnClose-nél.

Tipp

Ha kilépünk a Windows-ból, az OnCloseQuery esemény váltódik ki, és ezt felhasználhatjuk arra, hogy leállítsuk a kilépési folyamatot. Ebben az esetben az OnClose esemény még akkor sem következik be, ha az OnCloseQuery a CanClose paramétert igazra állítja.

Ha olyan alkalmazásokat készítünk, amelyeknek csak egyetlen főformja van, akkor az OnClose esemény kezelése nem jelent problémát. Ha új formokat adunk a programhoz, azok általában induláskor létrejönnek, és csupán a program terminálásakor kerülnek megsemmisítésre. Az ilyen esetekben van értelme a form elrejtésének. Ha azonban formjainkat dinamikusan hozzuk létre

(például azért, mert egyszerre több ugyanolyanra van szükségünk), akkor nem szabad elfelejtenünk megszabadulni tőlük bezárásukkor a bezáró eljárás caFree értékének használatával (a több formmal rendelkező alkalmazások készítéséről a 13. és 15. fejezetben szólunk majd részletesebben).

Most, hogy megbeszéltük a formok különleges tulajdonságait, egy igen fontos témára váltunk át: a form input és outputjára. Ha úgy döntünk, hogy nem (vagy csak korlátozott számban) használunk komponenseket, akkor is írhatunk bonyolult programokat, ha kezeljük a formhoz érkező eseményeket, és a form felületére közvetlenül rajzolunk. A következőkben tehát arról szó, hogy ezt hogyan is kell csinálni.

A billentyűzet-események figyelése

Általában a formok nem kezelik a hozzájuk érkező billentyűzeteseményeket közvetlenül. Ha a felhasználónak be kell gépelnie valamit, akkor azt egy szerkesztőmezőbe szokás megoldani. Ha billentyűparancsokat szeretnénk kezelni, akkor azokat a hozzájuk tartozó menükkel elintézhetjük, ahogy azt az utolsó fejezet HShort példájában is láthattuk (ott egy rejtett menüt használtunk a billentyűparancsok feldolgozására).

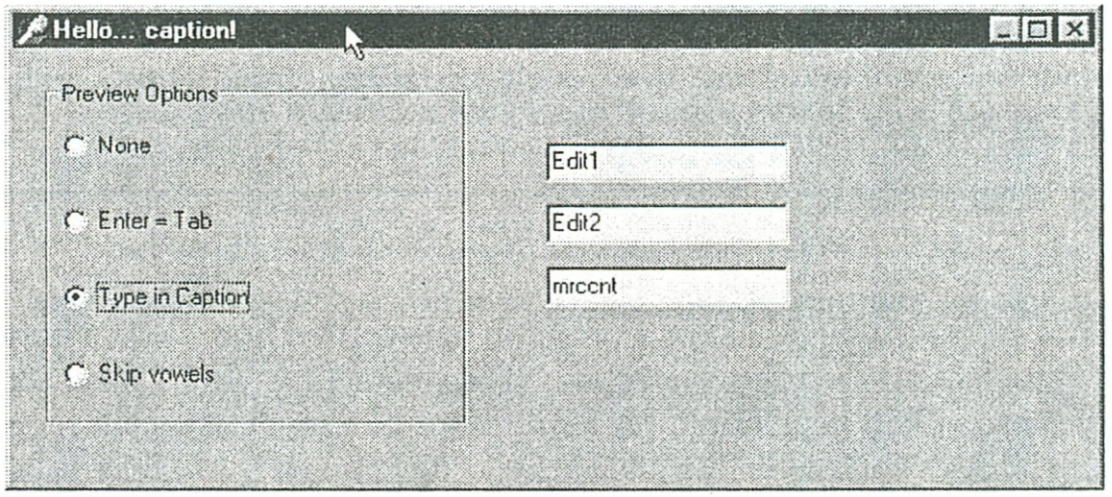
Persze előfordulhatnak olyan esetek, amikor valamilyen okból kezelni akarjuk a billentyűzet eseményeit. Ilyenkor állítsuk a form KeyPreview tulajdonságát igazra. Ezután – még ha vannak is vezérlőink – a form OnKeyPress eseménye mindig meghívódik. Az esemény ezután eléri a rendeltetési helyét, hacsak nem állítjuk meg úgy, hogy a küldött karakter értékét kinullázzuk (nem a 0 karakterrel, hanem a karakterkészlet 0 értékével, amit a #0 jelöl).

Az ide tartozó példa (melynek címe KPreview) nem rendelkezik semmilyen különleges tulajdonsággal (még a KeyPreview sincs igazra állítva). Csupán egy választógomb-csoport, és néhány szerkesztőmező található rajta (lásd 10.9. ábra):

```
object Form1: TForm1
  Caption = 'Key Preview'
  OnKeyPress = FormKeyPress
  object RadioPreview: TRadioGroup
    Caption = 'Preview Options'
    ItemIndex = 0
    Items.Strings = (
      'None'
      'Enter = Tab'
      'Type in Caption'
      'Skip vowels')
    OnClick = RadioPreviewClick
  end
  object Edit1: TEdit...
  object Edit2: TEdit...
```

Alapértelmezés szerint a program nem csinál semmit, egészen addig, amíg a választógombokkal be nem kapcsoljuk a billentyűzet eseményeinek követését:

```
procedure TForm1.RadioPreviewClick(Sender: TObject);
begin
  KeyPreview := RadioPreview.ItemIndex <> 0;
end;
```



10.9. ábra *A KPreview segítségével gépelhetünk a form címkéjébe*

Ezután elkezdjük kapni az OnKeyPress eseményeket, és elvégezhetjük a gombcsoport három különleges gombjainak valamelyike által megadott műveletet. Az, hogy melyik műveletet végezzük attól függ, hogy a csoport ItemIndex tulajdonsága milyen értéket mutat, ezért az eseménykezelő egy `case` elágazáson alapul:

```
procedure TForm1.FormKeyPress(Sender: TObject;
  var Key: Char);
begin
  case RadioPreview.ItemIndex of
    ...
```

Az első esetben, ahol a Key paraméter értéke #13 (ez az ENTER billentyűt jelenti), elfogjuk az üzenetet (a Key-t nullára állítjuk), és szimuláljuk a TAB billentyű lenyomását. Ezt sokféleképpen megtehetjük, én egy érdekes módot választottam: a formnak egy CM_DialogKey üzenetet küldtem, ahol átadom a tabulátor-billentyű kódját (VK_TAB):

```
1: //  Enter = Tab
       if Key = #13 then
       begin
```

```
    Key := #0;
    Perform (CM_DialogKey, VK_TAB, 0);
  end;
```

Megjegyzés

A CM_DialogKey üzenet egy nem dokumentált belső Delphi üzenet, aminek tárgyalása messze túlmutat könyvünk keretein. Többek között azonban ez is tárgyalásra kerül a Delphi Developer's Handbook című kötetben, amelyet társszerzőként írtam Tim Gooch-csal (Sybex, 1997).

Ha a form címsorát változtatjuk, akkor a program egyszerűen hozzáadja a beírt karaktert az aktuális Caption tulajdonsághoz, ahogy az a 10.9. ábrán is látszik. Két különleges esetet kell kezelnünk: ha a BACKSPACE billentyűt nyomjuk meg, akkor a string utolsó karakterét törölnünk kell (úgy, hogy a Caption tulajdonság minden karakterét bemásoljuk önmagába, kivéve az utolsót). Ha az ENTER-t nyomta meg a felhasználó, akkor a program leállítja ezt a műveletet úgy, hogy a gombcsoport ItemIndex tulajdonságát átállítja:

```
2: //  Type in Caption
  begin
    if Key = #8 then // backspace: remove last char
      Caption := Copy (Caption, 1,
        Length (Caption) - 1)
    else if Key = #13 then // enter: stop operation
      RadioPreview.ItemIndex := 0
    else // anything else: add character
      Caption := Caption + Key;
    Key := #0;
  end;
```

Végül, ha az utolsó lehetőséget választjuk, akkor a program leellenőrzi, hogy magánhangzót adtunk-e meg, és ha igen, akkor teljesen törli a beérkezett eseményt:

```
3: // Skip vowels
    if Key in ['a', 'e', 'i', 'o', 'u',
        'A', 'E', 'I', 'O', 'U'] then
      Key := #0;
```

Egéresemények figyelése

Minden alkalommal, amikor a felhasználó megnyomja az egér valamelyik gombját a formon, a Windows egy üzenetet küld az alkalmazásnak. A Delphi ezeket az üzeneteket eseményekként értelmezi, és futtatja a hozzájuk tartozó kódot. A két alapvető eseményünk a következő:

- OnMouseDown: ez akkor következik be, amikor az egér gombját lenyomja a felhasználó.
- OnMouseUp: ez akkor következik be, ha az egér gombját felengedték.

A rendszer egy másik fontos eleme az egér mozgásával kapcsolatos, az ide tartozó esemény az OnMouseMove. Ugyan ezt a három fogalmat – lenyomás, felengedés és mozgatás – könnyű megérteni, nem biztos hogy világos, miként kapcsolódnak ezek például az eddig is sokat használt OnClick eseményhez.

Már sokszor igénybe vettük a komponensek OnClick eseményét, de tartozik ilyen a formhoz is. Általában ez az esemény azt jelenti, hogy az egér bal gombja le lett nyomva és fel lett engedve ugyanazon az ablakon, vagy komponensen. Lehet azonban, hogy a lenyomás és felengedés között a felhasználó elmozgatja az egérmutatót a helyéről. Ha a gombot a mutató elhúzása után engedjük fel, akkor ez nem lesz kattintás esemény. Ilyenkor az ablak csak egy lenyomás, mozgatás, és egy felengedés üzenetet kap. Fontos az is, hogy kattintás üzenet csak az egér bal gombjától származhat.

Az egér gombjai

A PC-kkel használt legtöbb egér két, esetleg három gombbal rendelkezik. Általában ezekre a gombokra bal, középső és jobb gombokként hivatkozunk.

- A bal oldali egérgombot használjuk a legtöbbet. Ezzel választunk ki elemeket a képernyőről, kattintunk gombokon, választunk ki ás mozgatunk összetevőket, vagy aktiválunk dolgokat (általában dupla kattintással).
- A jobb oldali gombot előugró helyi menük előhívására használjuk. A múltban már sok különböző alkalmazás használta ezt a Windows 95-ben szabvánnyá vált módszert.
- A középső gombot csak nagyon ritkán használjuk, mert sok felhasználó egyszerűen nem rendelkezik vele, vagy nincs megfelelő meghajtóprogramja. Néhány CAD program használja ezt a gombot. Ha azt szeretnénk, hogy a programunk támogassa a középső gomb használatát, akkor ezt csak mint lehetőség kínáljuk fel a programunkban, különben kénytelenek leszünk ellátni felhasználóinkat háromgombos egerekkel, és a megfelelő meghajtóprogramokkal.

Ne feledjük, hogy a felhasználók felcserélhetik a jobb és a bal gomb szerepét, és a középső gomb szimpla kattintását alkalmazhatja a bal gomb dupla kattintása helyett. Amikor eseményeket kezelünk, nem a fizikai történés, hanem annak jelentése fontos számunkra.

A Windows használata egér nélkül

Most, hogy ehhez a részhez értünk, szólnom kell egy nagyon fontos dologról: a felhasználónak mindig meg kell adni a lehetőséget arra, hogy egér nélkül végezhesse el az általa kívánt műveleteket. Ez nem egy lehetőség: ez egy fontos szabály, amit ajánlatos mindig betartanunk. Persze sokszor egyszerűbb az alkalmazásokat egérrel kezelni, de erre soha ne kényszerítsük rá a felhasználót. Lehet, hogy felhasználóink egyáltalán nem is rendelkeznek egérrel helyhiány miatt, vagy mert éppen utaznak, hordozható számítógépet használnak, esetleg munkakörülményeik miatt.

Van másik oka is annak, hogy ajánlatos a billentyűzetet használnunk, ez pedig az, hogy bár az egérrel szépen és egyszerűen lehet dolgozni, a billentyűzettel sokkal gyorsabban kiadhatjuk ugyanazokat a parancsokat, mint egérrel. Ha például ügyesen és gyorsan tudunk gépelni, szövegszerkesztőnkben nem fogjuk az egeret használni mozgatásra vagy másolásra: egyszerűen a billentyűparancsokhoz fogunk folyamodni.

Ezekből az okokból kifolyólag fontos, hogy formunk komponenseinek tabulátor sorrendje helyesen be legyen állítva. Ne feledjük el a gombokhoz és menüparancsokhoz tartozó billentyűk beállítását sem. Esetleg kivételt képezhetnek ez alól a szabály alól a grafikus programok, de ne feledjük, hogy ha nehézkesen is, de még a Windows Paint programját is használhatjuk egér nélkül.

Az egéresemények paraméterei

Magam is egy grafikus programot készítettem példának, és most csak az egér eseményeivel fogunk foglalkozni. A Shapes című program első, minimális változatában az OnMouseDown eseménnyel fogunk foglalkozni. Az ide tartozó metódusnak több paramétere is van, ahogy azt a következő deklaráció mutatja:

```
procedure TShapesForm.FormMouseDown (
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

A szokásos Sender (küldő) paraméter mellett az alábbi négyet találjuk még itt:

- `Button`: ez jelzi, hogy a három egérgomb közül melyik lett lenyomva. A lehetséges értékek mbRight (jobb), mbLeft (bal) és mbCenter (középső). Ezek egymást kizáró értékek, hiszen a cél itt a lenyomott gomb meghatározása.

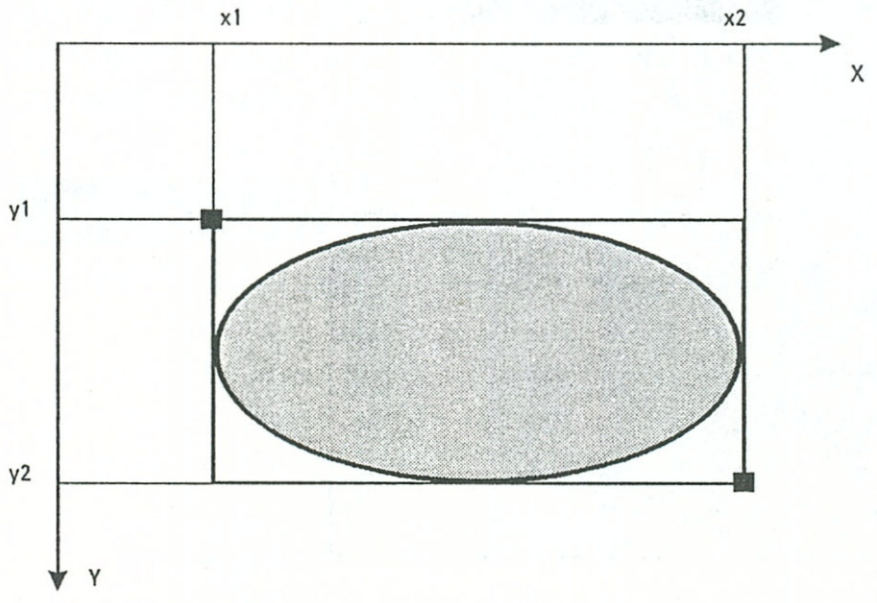
- `Shift`: azt jelzi, hogy milyen billentyűk voltak lenyomva az esemény bekövetkeztekor. Ezek az ALT, CTRL, SHIFT és maguk az egérgombok lehetnek. Ez a paraméter halmaz típusú, hiszen egyszerre több gomb is le lehet nyomva. Ez azt jelenti, hogy ezt a tulajdonságot nem egyenlőséggel, hanem halmazműveletekkel kell ellenőriznünk.
- Az `X` és az `Y` az egér pozícióját jelenti az esemény bekövetkeztekor, az aktuális ablak (form vagy vezérlő) kliens területéhez viszonyítva. A kezdőpont a kliens terület bal felső sarka.

Kattintás és rajzolás: a Shapes1 példa

Elkezdhetjük hát a Shapes1 című példa elkészítését, amelyben egy egyszerű OnMouseDown eseményt fogunk használni. Az osztály nevéből is kiderül, hogy a használt form (amelyen egyébként nincsenek komponensek) nevét ShapesForm-ra változtattam:

```
procedure TShapesForm.FormMouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
    Canvas.Ellipse (X-10, Y-10, X+10, Y+10);
end;
```

A program egy kört rajzol a formra, ha a felhasználó lenyomja rajta a gombot. A kör közepe a kattintás helye lesz. A kör megrajzolásához az Ellipse (ellipszis) eljárást használjuk, mivel sem a Windows-nak, sem pedig a Delphi-nek nincsen kört rajzoló eljárása. Az Ellipse metódus négy paramétert használ, ezek az ellipszist körülvevő téglalapot adják meg, a 10.10. ábrán látható módon.



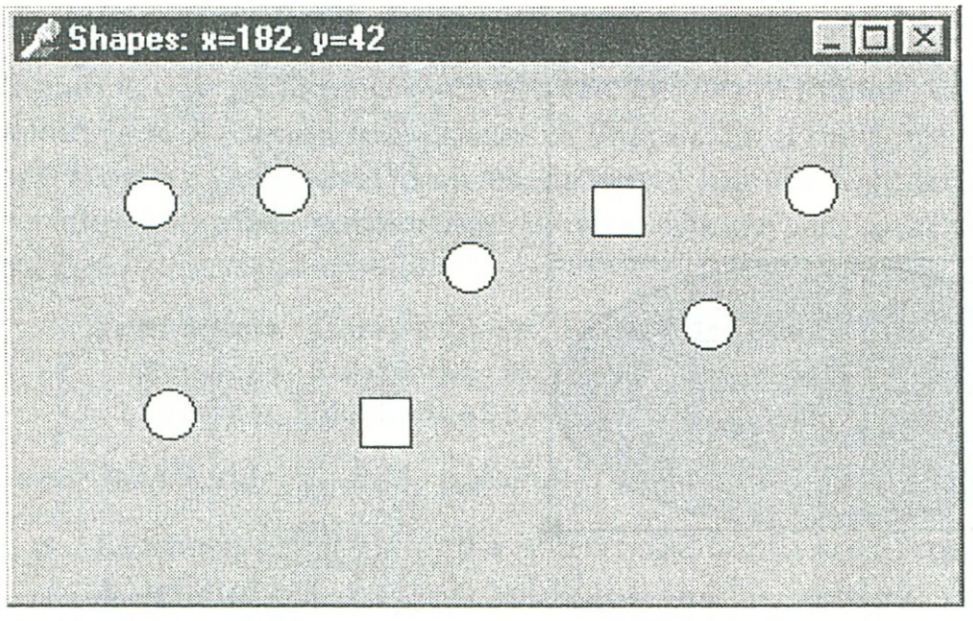
10.10. ábra *Az ellipszist tartalmazó téglalap*

A formra rajzoláshoz egy speciális, Canvas (vászon) nevű tulajdonságot fogunk használni. A TCanvas osztálynak két fontos tulajdonsága van: egyrészt rajzolásra alkalmas eszközökkel rendelkezik (például ecsettel, tollal és fontokkal), másrészt olyan metódusokat ad a kezünkbe, melyekkel a rajzoló eszközöket használhatjuk. A TCanvas osztályról részletesebben is hallhatunk a következő fejezetben.

Adjunk néhány új tulajdonságot a programhoz, mielőtt még belemerülnénk a rajzolás rejtelmeibe. Először is lehetővé akarom tenni, hogy attól függően, milyen billentyűt nyomtunk le az egér bal gombjával együtt, más formák jelenjenek meg a képernyőn. Ezt az OnMouseDown esemény harmadik paraméterének használatával érhetjük el:

```
procedure TShapesForm.FormMouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
    if ssShift in Shift then
      Canvas.Rectangle (X-10, Y-10, X+10, Y+10)
    else
      Canvas.Ellipse (X-10, Y-10, X+10, Y+10);
end;
```

Ha most a felhasználó lenyomja a SHIFT billentyűt és megnyomja az egér bal gombját, a program egy kis négyzetet rajzol kör helyett (lásd 10.11. ábra). Továbbfejleszthetjük a példát úgy, hogy vonalakat, vagy lekerekített sarkú négyzeteket rajzoljon. A Delphi Súgójában a TCanvas osztály leírásánál megtalálhatjuk az ide vonatkozó metódusokat.



10.11 ábra *A Shapes1 példa futás közben*

Még egy dolgot kell változtatnunk a Shapes példán, és készen leszünk vele. Azt szeretném, hogy az egérmutató kliens pozíciója megjelenjen a képernyőn, így pontosabban lehessen rajzolni. Mindig, amikor az egér áthalad a form felett, egy OnMouseMove esemény generálódik. Az ehhez az eseményhez tartozó kódrészletben szöveggé alakíthatjuk az egér X és Y koordinátáit, és megjeleníthetjük azokat. A form nem rendelkezik komponensekkel, így annak címkéjét fogjuk az értékek megjelenítésére használni (ez látszik a 10.11. ábra ablakán is).

```
procedure TShapesForm.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  Caption := Format ('Shapes: x=%d, y=%d', [X, Y]);
end;
```

Most már egy kicsit érdekesebb a programunk. A példa lényege azonban nem is a rajzolás, hanem hogy megértsük, hogyan működik az egér. Próbáljuk ki a következőt: futtassuk a programot, majd mozgassuk az ablakot olyan pozícióba, hogy ne legyen aktív, de a címsora látszódjon. Ha most a form fölé mozgatjuk az egérmutatót, látni fogjuk, hogy a koordináták így is változnak. Ez azt jelenti, hogy az OnMouseEvent eseményt akkor is megkapja alkalmazásunk, ha az nem aktív, és bizonyítja egy korábban már említett állításomat, miszerint az egérüzeneteket – néhány kivételtől eltekintve – az mutató alatti ablak kapja meg.

Rajzolás a formra

A Shapes1 programmal köröket és négyzeteket rajzolhatunk formunk felületére, ahogy azt a 10.11. ábra is mutatja. Úgy tűnik, a program helyesen működik, de ez sajnos nem így van. Több gond is van vele: ha a form egy részét eltakarjuk egy másik ablakkal, akkor a letakart részre rajzolt dolgok eltűnnek. Ugyanez történik akkor, ha kissebre vesszük formunkat. Játsszunk egy kicsit a programmal, és hamar rájövünk, mire is gondolok.

Megjegyzés

Miért nem tárolja a Windows minden ablakát egy bittérképen? Nos, a válasz egyszerű: azért, hogy takarékoskodjon a memóriával. Egy 256 színű 300 × 400 pixel méretű bittérkép körülbelül 120 KB memóriát foglal el. Ha növeljük a pixelenkénti színek számát, a teljes méretű bittérképek gyorsan elérhetik az 1 MB méretet, vagy 1280 × 1024 felbontásnál, 16 millió színnél a 4 MB-ot. Ha ez lenne az alapértelmezés szerinti ablakkezelés, akkor néhány egyszerű alkalmazás futtatása legalább 8, de inkább 16 MB memóriát igényelne.

Mindez a Windows természete miatt van így. Ha rajzolunk egy ablakra, a Windows nem tárolja az elkészített képet, így ha a felületet eltakarjuk, a rajzok elvesznek.

Több technikát is ismerünk arra, hogy a form tartalmát egy bittérképben tároljuk. A Windows-ban ezek külön kóddal, a Delphi-ben pedig a megfelelő komponensek használatával oldható meg. Ebben a fejezetben a közvetlen outputtal fogunk foglalkozni, és a szokásos, kifestésen alapuló Windows technikát fogom alkalmazni (erről lesz szó a most következő részben). Az ez utáni fejezetben azt fogom megmutatni, hogyan lehet Image (kép) komponenst használni az ablak tartalmának tárolására.

A rajzeszközök

Egyelőre fejlesszük tovább egy kicsit programunkat a Canvas eszközeinek felhasználásával. Íme ezeknek az eszközöknek (amelyek egyébként GDI (Graphics Device Interface) objektumok a Windows rendszerkönyvtárából) a listája:

- A Brush (ecset) tulajdonság határozza meg a zárt felületek kitöltési színét. Ezt használjuk zárt alakzatok, körök, vagy téglalapok kifestésénél. Az ecset tulajdonságai közé tartozik a Color (szín), Style (stílus), és a Bitmap (bittérkép).
- A Pen (toll) tulajdonság határozza meg a rajzolt vonalak vastagságát és színét. Az ide tartozó paraméterek a Color (szín), Width (szélesség), és a Style (stílus), amely több különböző pontozott és szaggatott típust tartalmaz (ezeket csak egy pixel vastagságú vonalaknál használhatjuk).
- A Font tulajdonság a Canvas objektum TextOut metódusával kiírt szövegek betűtípusát határozza meg. Egy betűtípusnak van neve (Name), mérete (Size), stílusa (Style), színe (Color), és így tovább.

Az ecsetek, tollak és fontok (de a formok, és más komponensek is) rendelkeznek a Color tulajdonsággal. Ha azonban egy elem színét meg akarjuk változtatni, ismernünk kell, hogy a Windows hogyan kezeli a színeket. Elméletileg a Windows 24 bites RGB színeket használ, ami azt jelenti, hogy a kívánt színt a három alapszín 0 és 255 közötti értékével tudjuk beállítani, így több mint 16 millió színt tudunk használni.

Sajnos azonban ezt a teljesítményt nem minden monitorkártya képes nyújtani. Az ilyen esetekben a Windows megpróbálja kikeverni a használni kívánt színt az előállíthatókból, vagy a hozzá legközelebb álló megjeleníthetőt alkalmazza. Ecsetek színéhez, és a formok hátterének kifestéséhez a rendszer mindig színkeverést, tollak használatakor pedig közelítéses technikát használ.

A tollak helyzetét lekérdezhetjük (de nem változtathatjuk) a Canvas PenPos tulajdonságával. A toll helyzete határozza meg azt, hogy honnan fog indulni a LineTo metódus által húzott vonal. A toll pozícióját a MoveTo metódussal állíthatjuk be. Érdekesek lehetnek még a vonalakat és színeket megadó tulajdonságok, a CopyMode és ScaleMode értékei, valamint a Pixels tulajdonság, amellyel közvetlenül megadhatjuk az egyes pixelek színét a form felületén.

Alakzatok rajzolása

Most a Shapes programot fogjuk továbbfejleszteni. Egy menü segítségével fogjuk a rajzolt alakzatok formáját és a használt toll és ecset színét, valamint a vonal vastagságát kiválasztani. Az ezekhez szükséges kód egy kicsit hosszabb a megszokottnál, de igen egyszerű. Íme a menü szerkezete, ami a form szöveges leírásából való:

```
object MainMenu1: TMainMenu
  object File1: TMenuItem
    object New1: TMenuItem
    object N1: TMenuItem
    object Exit1: TMenuItem
  object Colors1: TMenuItem
    object PenColor1: TMenuItem
    object BrushColor1: TMenuItem
  object Size1: TMenuItem
    object IncreasePenSize1: TMenuItem
    object DecreasePenSize1: TMenuItem
    object N2: TMenuItem
    object IncreaseShapeSize1: TMenuItem
    object DecreaseShapeSize1: TMenuItem
  object Help1: TMenuItem
    object AboutShapes1: TMenuItem
```

A form beállítja és eltárolja a kör aktuális sugarának méretét (ami megegyezik a négyzet oldalának felével):

```
private
  Radius: Integer;
```

Az összes többi, menüparancs által beállított értéket a form Canvas tulajdonságában tároljuk. A sugár (melynek kezdeti értéke 5) növelése és csökkentése elég egyszerű. Az egyetlen érdekesebb rész itt a Decrease (csökkentés) menü kikapcsolása, amelyet azért kell megtennünk, hogy a sugár értéke ne csökkenjen nulla alá.

```
procedure TShapesForm.IncreaseShapeSize1Click(Sender:
    TObject);
begin
  Radius := Radius + 5;
  DecreaseShapeSize1.Enabled := True;
```

```
end;

procedure TShapesForm.DecreaseShapeSize1Click(Sender:
    TObject);
begin
  Radius := Radius - 5;
  if Radius < 10 then
    DecreaseShapeSize1.Enabled := False;
end;
```

Ha akarjuk, elvégezhetünk hasonló ellenőrzéseket a növelésnél is. A toll méretének megváltoztatására használt kód is ehhez hasonló, azzal a különbséggel, hogy a Canvas Pen elemének Width (szélesség) tulajdonságán alapul:

```
procedure TShapesForm.IncreasePenSize1Click(Sender:
    TObject);
begin
  Canvas.Pen.Width := Canvas.Pen.Width + 2;
  DecreasePenSize1.Enabled := True;
end;

procedure TShapesForm.DecreasePenSize1Click(Sender:
    TObject);
begin
  Canvas.Pen.Width := Canvas.Pen.Width - 2;
  if Canvas.Pen.Width < 3 then
    DecreasePenSize1.Enabled := False;
end;
```

A határok (vagyis a toll) és a felszínek (vagyis az ecset) színének beállításához a standard Color párbeszédablakot használtam. Íme a két ide tartozó metódus:

```
procedure TShapesForm.PenColor1Click(Sender: TObject);
begin
  ColorDialog1.Color := Canvas.Pen.Color;
  if ColorDialog1.Execute then
    Canvas.Pen.Color := ColorDialog1.Color;
end;
```

Mindezt az információt a program – közvetlen vagy közvetett módon – új alakzatok rajzolásakor használja fel:

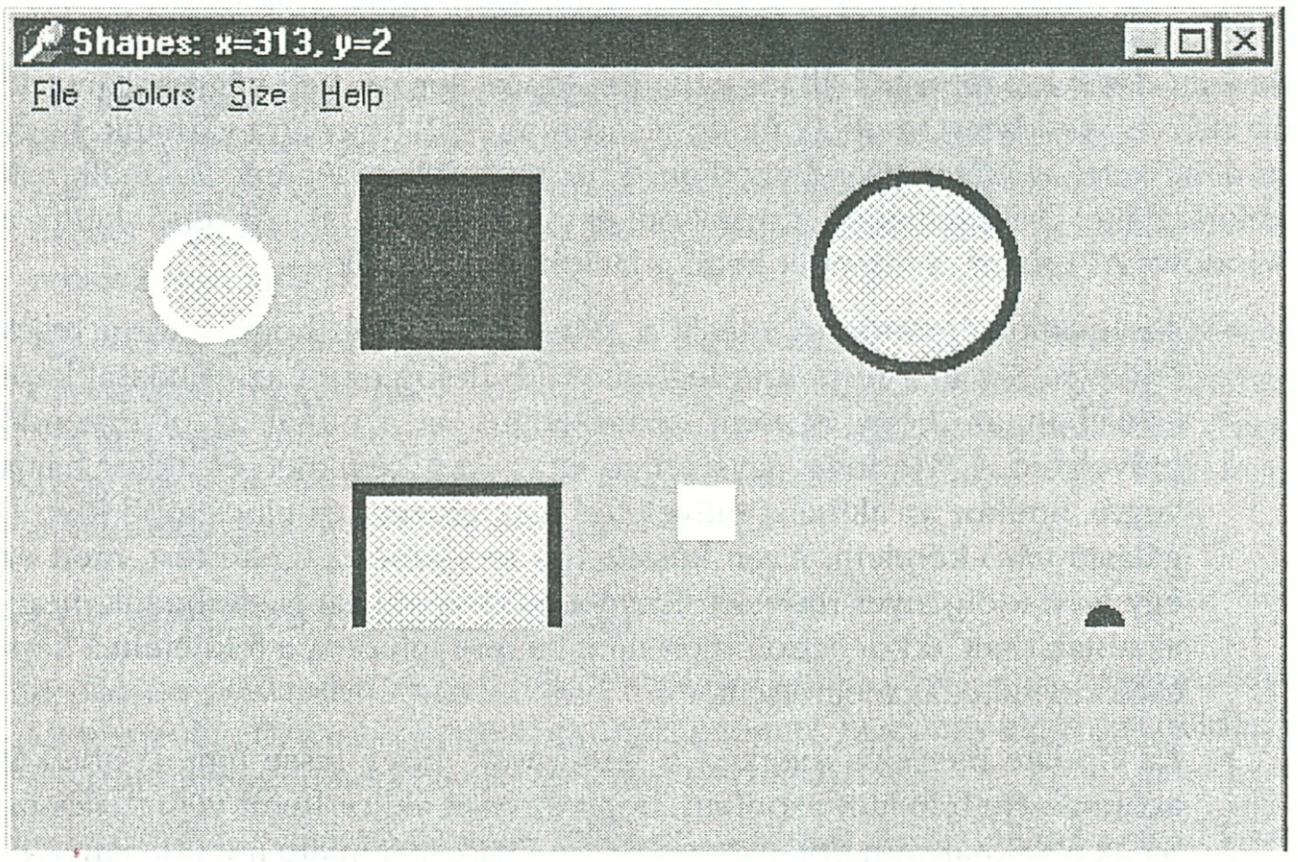
```
procedure TShapesForm.FormMouseDown(Sender: TObject;
  Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
    if ssShift in Shift then
      Canvas.Rectangle (X - Radius, Y - Radius,
        X + Radius, Y + Radius)
```

```
    else
      Canvas.Ellipse (X - Radius, Y - Radius,
        X + Radius, Y + Radius);
end;
```

A kód egyetlen igazán furcsa része az a metódus, amelyik a **File | New** menüpont kiválasztásakor törli az ablak tartalmát:

```
procedure TShapesForm.New1Click(Sender: TObject);
begin
  Repaint;
end;
```

A kód egyszerűen meghívja a form Repaint (újrafestés) metódusát. Ez újrafesti az ablak teljes felületét, kitörölve annak tartalmát. Természetesen ez azt jelenti, hogy mindent, amit addig rajzoltunk, elveszítünk, hiszen nem tároltuk el sehol. Ez a program is ugyanazzal a nagy gonddal küzd, mint az előző (ahogy ezt a 10.12. ábrán láthatjuk is). Ha egy másik ablakot mozgatunk fölé, vagy átméretezzük a formot, akkor elveszítjük a tartalmát. Szükségünk van tehát egy módszerre, amely segítségével tárolhatjuk az ablak tartalmát, és újrafesthetjük, ha szükség van rá.



10.12. ábra *A Shapes2 program jobban működik elődjénél, de még mindig nem tökéletes, ahogy azt az ablak jobb oldala is mutatja*

Rajzolás és festés a Windows-ban

Itt az ideje, hogy szembenézzünk a Windows rajzolási és festési problémáival. Mit is értünk ezek alatt pontosan?

- Rajzolás (drawing) alatt azt értjük, hogy meghívjuk a Canvas metódusait, ahogy azt a Shape programban is tettük. Itt a kép elveszhetett, mivel nem volt tárolva sehol, ahogy a 10.12. ábrán láttuk. Ebből kifolyólag az alkalmazás sem tudja helyesen újrafesteni magát.
- Festést (painting) akkor végzünk, amikor tudatjuk az alkalmazással, hogy adott körülmények között hogyan kell újrafestenie a teljes felületét. Ha megadjuk, hogy hogyan kell az ablakot újrafesteni, akkor ez a metódus automatikusan le fog futni minden olyan alkalommal, amikor a formot meg kell jeleníteni, ezúttal helyesen.

Általában Windows programok a festést használják megjelenítéshez, leszámítva azt a néhány grafikus programot, amelyek a form teljes képét egy bittérképen ábrázolják (erre látunk példát a következő fejezetben). A most leírásra kerülő technikának beszédes neve van, úgy hívják, hogy „tárolás és festés". Amikor a felhasználó lenyom egy egérgombot, vagy más hasonló műveletet végez, el kell tárolnunk az eseményhez kapcsolódó paramétereket, majd a festő metódusban a tárolt információ segítségével meg kell festenünk a megfelelő képet.

Ez a módszer két lépésből áll, és ezt a két lépést sorban kell végrehajtanunk, megkérve a rendszert az ablak újrafestésére – anélkül hogy arra várnánk, hogy ez a művelet magától végrehajtódjon. Erre több lehetőségünk is kínálkozik (mindegyiket használhatjuk formokon és vezérlőkön is). Az első kettő a Windows API része, a második kettő a Delphi-hez tarozik:

- Az Invalidate metódus értesíti a Windows-t arról, hogy a form teljes felületét újra kell festeni. A legfontosabb dolog ezzel az eljárással kapcsolatban az, hogy itt nem kényszerítjük ki azonnal az elvégzendő műveletet. A Windows egyszerűen eltárolja a kérelmet, és akkor hajtja végre, amikor az aktuális művelettel már végzett, és nincs több kiszolgálásra váró kérelem. Azért késlelteti a rendszer az újrafestést, mert ez egy igen időigényes művelet. Ezzel a módszerrel azt is elérhetjük, hogy az ablak csak akkor rajzolódjon újra, ha már minden, a felületéhez kapcsolódó változás megtörtént.
- Az Update metódus megkéri a Windows-t, hogy fesse újra az ablakot azonnal. Ne feledjük azonban, hogy ez csak akkor kerül végrehajtásra, ha az ablaknak van érvénytelen része. Ez az Invalidate metódus meghívásakor, vagy a felhasználó tényked ésének eredményeképp következhet be. Ha nincs érvénytelen területünk, a metódus nem csinál semmit.

Ez az oka annak, hogy az Update és az Invalidate metódusokat általában együtt használják. Pontosan így működnek egyébként a Delphi Repaint és Refresh (frissítés) metódusai is.

- A Repaint metódust meghívja az Invalidate-et és az Update-et egymás után, ez pedig azonnal kiváltja az OnPaint eseményt. A Refresh működése ehhez igen hasonló, csupán a komponenseknél mutat egy kis eltérést.

Tipp

A Repaint és Refresh metódusok tökéletesen felcserélhetők egymással. A Borland a Delphi 1 Súgójában még azt állította, hogy különböznek egymástól, de ez nem igaz. A dokumentáció által mondottak helyett, ha nem akarjuk törölni a form (vagy a használt komponens) hátterét, akkor a nem dokumentált ControlStyle tulajdonság csOpaque jelzőjét kell igazra állítani.

Ha azt szeretnénk, hogy formunkat újrafesse a Windows, akkor általában az Invalidate metódust érdemes meghívnunk. Ez akkor különösen fontos, ha sűrűn kell végrehajtanunk ezt a műveletet, mert ha a rendszernek túl sok időbe telik az újrafestési kérések teljesítése, akkor a többszörös kéréseket csak egyszer hajtja végre. A Windows wm_Paint üzenete egyfajta alacsony prioritású üzenet, ami azt jelenti, hogy más üzenetek és műveletek megelőzik ezt a végrehajtás sorrendjében.

Ha viszont a Repaint metódust hívjuk meg sokszor, akkor a képernyőt – más üzenetektől függetlenül – sokszor újra kell festeni, és az ilyen jellegű műveletek lassúsága miatt alkalmazásunk kevésbé fog hatékonyan működni. Természetesen előfordulhatnak olyan esetek, amikor azonnal újra kell festeni ablakunkat – az ilyen esetekre való a Repaint.

Egy egyszerű alakzat festése

A Shapes példa első verziója csak kitöltés nélküli köröket és négyzeteket tudott rajzolni. Újra ezt a programot készítettem el, hogy egy egyszerű példán keresztül mutassam be a tárolás és festés technikáját.

A bal egérgomb lenyomásakor három értéket kell tárolnunk: az alakzat középpontjának koordinátáit, és azt a logikai értéket, amely megadja, hogy kört, vagy négyzetet rajzolunk-e. A Shapes3 példa formja ezért két új lokális mezőt is tartalmaz:

```
type
  TShapesForm = class(TForm)
    ..
  private
    Center: TPoint;
    Circle: Boolean;
  end;
```

Ezeket az értékeket a FormMouseDown eljárással állítjuk be, amely meghívja az Invalidate metódust:

```
procedure TShapesForm.FormMouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
    // store the center and the kind of shape
    Center.X := X;
    Center.Y := Y;
    Circle := not (ssShift in Shift);
    // ask to repaint the form
    Invalidate;
  end;
end;
```

Az Invalidate meghívása közvetetten aktiválja a FormPaint metódust, amit a form OnPaint eseményéhez társítottunk. Ez az aktuális értékekkel kirajzol egy alakzatot:

```
procedure TShapesForm.FormPaint(Sender: TObject);
begin
  if Circle then
     Canvas.Ellipse (Center.X - 10, Center.Y - 10,
       Center.X + 10, Center.Y + 10)
  else
     Canvas.Rectangle (Center.X - 10, Center.Y - 10,
       Center.X + 10, Center.Y + 10);
end;
```

A privát mezők kezdeti értékeit még a program elején, a FormCreate metódussal beállítjuk. A kezdeti koordináták negatívra vannak állítva, így először az alakzat nem látható, bár az OnPaint metódusa már induláskor is lefut. Ahogy az a program futtatásakor láthatjuk, most csak egyetlen alakzatot tudunk egyszerre a képernyőre rajzolni. A Shapes példa utolsó változata igen nagyra sikeredett, de ez nem jelenti azt, hogy a lehető legjobban működne. Természetesen összefűzhetjük ezt a verziót a Shapes2-vel, és így az alakzatok színét és méretét is meghatározhatjuk.

Több alakzat festése

Egyszerűen meg lehet oldani több elem tárolását a Delphi segítségével: egy TList objektumot kell alkalmaznunk, amelyben tetszőleges típusú objektumokat tárolhatunk. Ezt használtam én is a Shapes4 példában, amely az alkalmazás második és harmadik változatának egyvelege. Itt ugyanazt a formot és menüszerkezetet használjuk, amit a Shapes2-nél.

A programban egy saját TShapeData nevű adattípusban tároljuk az alakzatok attribútumait:

```
type
  TShapeData = class
    Circle: Boolean;
    X, Y, Size, PenSize: Integer;
    PenColor, BrushColor: TColor;
  end;
```

A form rendelkezik egy ShapesList nevű TList osztályú objektummal, amelyet az OnCreate esemény kezelésénél készítünk el:

```
procedure TShapesForm.FormCreate(Sender: TObject);
begin
  Radius := 5;
  ShapesList := TList.Create;
end;
```

A program minden olyan alkalommal hozzáad egy új objektumot a listához, amikor a bal gombot megnyomja a felhasználó:

```
procedure TShapesForm.FormMouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  Shape: TShapeData;
  InvRect: TRect;
begin
  if Button = mbLeft then
  begin
    Shape := TShapeData.Create;
    if ssShift in Shift then
      Shape.Circle := False
    else
      Shape.Circle := True;
    Shape.X := X;
    Shape.Y := Y;
    Shape.Size := Radius;
    Shape.PenSize := Canvas.Pen.Width;
    Shape.PenColor := Canvas.Pen.Color;
    Shape.BrushColor := Canvas.Brush.Color;
```

```
      ShapesList.Add (Shape);
      Invalidate;
    end;
  end;
```

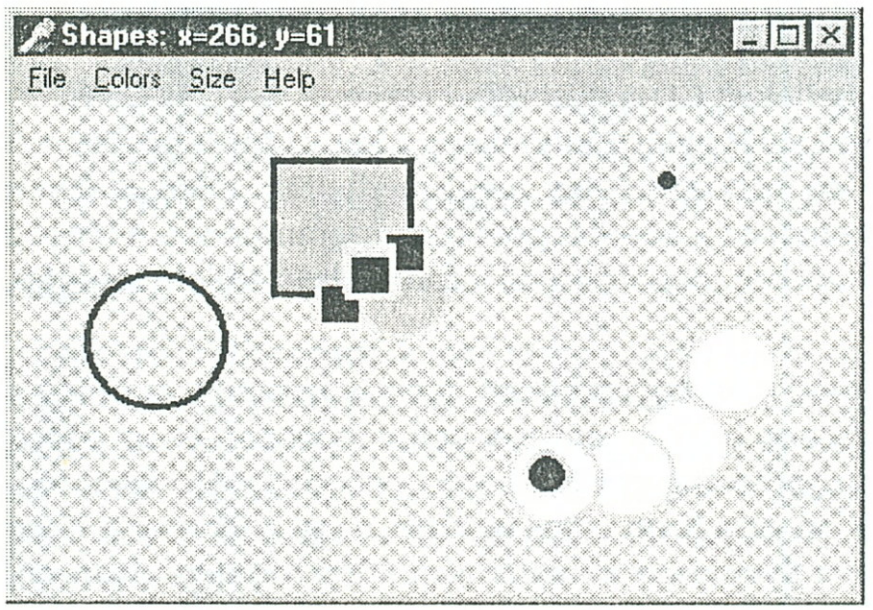
Az OnPaint eseményhez tartozó metódusban a lista minden elemét újrafesti a rendszer. Ez a kódrészlet hatással van a Canvas tulajdonságaira is, így tárolnunk kell annak értékeit, majd visszaállítani a metódus végén. Ezt azért kell megtennünk, mert ha a felhasználó megváltoztatja az utolsó alakzat színét, és nem mentjük és állítjuk vissza az aktuális színt, akkor a Canvas az utolsó alakzat attribútumait fogja örökölni:

```
procedure TShapesForm.FormPaint(Sender: TObject);
var
  I, OldPenW: Integer;
  CurShape: TShapeData;
  OldPenCol, OldBrushCol: TColor;
begin
  // store the current Canvas attributes
  OldPenCol := Canvas.Pen.Color;
  OldPenW := Canvas.Pen.Width;
  OldBrushCol := Canvas.Brush.Color;

  // repaint each shape in the list
  for I := 0 to ShapesList.Count - 1 do
  begin
    CurShape := ShapesList.Items [I];
    with CurShape do
    begin
      Canvas.Pen.Color := PenColor;
      Canvas.Pen.Width := PenSize;
      Canvas.Brush.Color := BrushColor;
      if Circle then
        Canvas.Ellipse (X-Size, Y-Size, X+Size, Y+Size)
      else
        Canvas.Rectangle (X-Size, Y-Size,
            X+Size, Y+Size);
    end;
  end;

  // reset the current Canvas attributes
  Canvas.Pen.Color := OldPenCol;
  Canvas.Pen.Width := OldPenW;
  Canvas.Brush.Color := OldBrushCol;
end;
```

A TShapeForm osztály több metódusa egyszerű. A program leellenőrzi, hogy valóban ki akarunk-e lépni belőle, vagy hogy törölni akarjuk-e az alakzatok listáját (a **File | New** paranccsal):

```
procedure TShapesForm.New1Click(Sender: TObject);
begin
  if (ShapesList.Count > 0) and (MessageDlg (
    'Are you sure you want to delete all the shapes?',
    mtConfirmation, [mbYes, mbNo], 0) = idYes) then
  begin
    ShapesList.Clear;
    Refresh;
  end;
end;
```

Így már szép lesz a programunk (lásd 10.13. ábra), és bármi is történik az ablakkal, az alakzatok a helyükön maradnak.



10.13. ábra *A Shapes4 példa futás közben*

Egyetlen hátránya a programnak, hogy minden új alakzat hozzáadásakor újra kell rajzolnunk minden meglevőt, amitől a képernyő zavaróan villog. Ahelyett, hogy érvényteleníténénk, törölnénk, majd újrarajzolnánk az egész felületet, megtehetjük azt is, hogy ezt az ablak egy kisebb részén végezzük el. Ehhez a Delphi nem tartalmaz semmilyen különleges eszközt, de meghívhatjuk a Windows API InvalidateRect függvényét:

```
procedure InvalidateRect(Wnd: HWnd;
  Rect: PRect; Erase: Bool);
```

A három paraméter az ablak kezelőjére (ami a form Handle tulajdonsága), a kifestendő téglalapra, és arra vonatkozik, hogy törlődjön-e a terület tartalma újrafestés előtt. A Shapes4 példa FormMouseDown eljárásának Invalidate hívását a következőkre cseréljük:

```
var
  InvRect: TRect;
begin
  ...
  {standard code:}
  // Invalidate;

  {optimized code:}
  InvRect := Rect (X - Radius - Shape.PenSize,
  Y - Radius - Shape.PenSize, X + Radius +
Shape.PenSize,
  Y + Radius + Shape.PenSize);
  InvalidateRect (Handle, @InvRect, False);
end;
```

A program megtalálható a mellékelt CD-n. Kitörölhetjük az új részt, és érvényesíthetjük a régit, így világossá válik majd a különbség a két változat között. Azt ajánlom, kísérletezzünk a két változattal, így megismerhetjük, miért is olyan fontos az InvalidateRect használata a lassabb grafikus programoknál. A lényeg az, hogy a Repaint és az Invalidate meghívásánál a form teljes felülete törlődik, majd újra kirajzolódik.

Az ablak egy kisebb területének érvénytelenítése azért is hasznos, mert így a kép kevésbé fog villogni, hiszen kisebb területet kell változtatni, és ez rövidebb ideig is tart. Valójában egyébként minden festési művelet valamilyen téglalapra korlátozódik, általában ez a form teljes területe, vagy a mi esetünkben a megadott rész. A Windows ilyenkor automatikusan átlép minden olyan rajzolási műveletet, amely a megadott téglalapon kívül esik, így is időt takarítva meg.

Ha pontosak akarunk lenni, akkor a vágási téglalap helyett a vágási terület kifejezést kell használnunk, mert ez a régió bármilyen komplex formát felvehet. Ha további részletekre vagyunk kíváncsiak, utánanézhetünk az InvalidateRegion és CreateRegion függvényeknek a Windows Súgójában.

Tipp

Az InvalidateRect hívással érvénytelenített területnek elég nagynak kell ahhoz lenni, hogy az utolsó alakzat – az esetleges vastag keretével együtt (hacsak nem psInsideFrame tollstílust használunk) – beleférjen. Ez az oka annak, hogy ehhez hozzáadtam a toll méretét.

A Delphi kimeneti komponensei

A Shapes példa négy verzióját is elkészítettük, szinte minden komponens használata nélkül. Másik megoldásként alkalmazhattuk volna a Delphi PaintBox és Shape összetevőit. Mire jó vajon ez a két eszköz?

- A PaintBox komponenst akkor használjuk, amikor a form egy adott területére kell festenünk, és ez a terület mozoghat a formon belül. Például egy párbeszédablakot gond nélkül újrafesthetünk ezzel a metódussal, nem kell attól félnünk, hogy a megjelenítés keveredik más vezérlőkkel. Használhatjuk egymás mellett lévő komponensek (például eszköztárak) megjelenítésére is. A Shapes példában nem lett volna értelme ezt használni, hiszen ott mindig az egész felületet újrafestettük.
- A Shape komponenssel alakzatokat festhetünk a képernyőre pontosan úgy, ahogy eddig csináltuk. A példáinkban is használhattuk volna ezt az összetevőt, de meg akartam mutatni, hogyan lehet közvetlenül rajzolni az ablakokra. Saját módszerünk nem sokkal volt bonyolultabb a Delphi által sugallttól. A Shapes komponens használatával továbbfejleszthetjük programunkat: lehetővé tehetjük, hogy a felhasználó mozgassa és törölje a kész alakzatokat. A Shape komponens használatára a 15. fejezet MdiDemo4 programjában láthatunk példát.

Hogyan tovább?

Az első fejezet óta tudjuk már, hogy minden form egy ablak, de mostanáig nem foglalkoztunk a formok tulajdonságaival. Most megtanultuk, hogyan változtathatjuk méretét és helyzetét, hogyan szerezzük meg az egér- és billentyűzeteseményeket, és hogyan rajzoljunk felületére. Megemlítettünk két globális objektumot, a Screen-t és az Application-t, amelyekről a 25. fejezetben lesz szó részletesen.

A könyv további fejezetei a formokkal fog foglalkozni. A 12. és 15. fejezetben megnézzük az eszköztárak és párbeszédablakok készítését, a többdokumentumos alkalmazásokat, a görgetősávok, többoldalas formok, megosztott ablakok, és MDI alkalmazások használatát. Ebből a felsorolásból is látható, hogy a formok központi szerepet játszanak a Delphi programozásban, és sok fontos dolog van még, amiről nem ejtettünk szót velük kapcsolatban.

Mielőtt azonban belevágnánk ezekbe a témákba, a következő fejezetben megnézzük, hogyan használjunk bizonyos komponenseket, olyanokat, amikről a fejezet utolsó részében esett szó, és olyanokat, mint a gombok, listapanelek, és rácsok (soha nem mondtam, hogy meg lehet tanulni Delphiben programozni két nap alatt, de ha rászánjuk az időt – és tovább olvassuk a könyvet – elsajátíthatjuk a téma minden csínját-bínját).

11. FEJEZET

Grafikus komponensek

E fejezet tartalma:

- A felhasználói felület grafikus fejlesztése
- Bitmap a gombon
- Változó kép a gombon
- Az Animáció kontrol
- Képmegjelenítő
- Rajzolás bittérképbe
- Grafikus listák
- Grafikus rácsok
- A TeeChart használata
- A Dátum-Idő kiválasztó kontrol

Grafikus komponensek

A Windows alapvető kontroljai mellett a Delphi több fejlett, modern felhasználói felülettel rendelkező komponenst is tartalmaz. Néhány ezek közül a szokásos komponensek grafikus változatai (mint például a gombok és lista dobozok), de vannak teljesen újak is. A Windows 95 új, általános kontroljai mellett Delphi által definiáltak is vannak, amelyeket a 16-bites Borland fejlesztői környezetekben is megtalálhatunk. Ezekről az új, fejlett vezérlőkről lesz szó ebben, és még néhány ezt követő fejezetben.

Az előző fejezetben láthattuk, hogy könnyedén rajzolhatunk egy form felszínére. Ugyanezt más komponensekkel is megtehetjük, bár azt már tudjuk, hogy a Windows kirajzolási modellje nem éppen egyszerű. Ezért előnyösebb az előre elkészített grafikus kontrolok használata. Az alkalmazásunk felhasználói felületét gyorsan, és könnyen fejleszthetjük grafikus gombokkal, táblázatokkal, vázlatokkal, és ugyanakkor rendkívül hasznos funkciókkal is kibővítjük az alkalmazást.

A felhasználói felület grafikus fejlesztése

A korai Windows alkalmazások kinézetre sokban különböznek a maiaktól. A Windows 3.1 megjelenésével általánossá vált az eszköztárak, grafikus gombok, grafikus listapanelek, és még sok más használata a felhasználói felületek kialakításához. A 16-bites Windows API közvetlenül nem támogatta ezeket az elemeket, és használatuk sok programozói munkát igényelt. A dolgok mostanra már megváltoztak. A Windows 95-ben van több fejlett grafikus kontrol, amelyeket az alkalmazások használhatnak.

Röviden leírunk néhányat ezek közül, és egyszerű példákban bemutatjuk a működésüket, ahogy szoktuk. Mielőtt nekikezdenénk, vegyük figyelembe, hogy a grafikus elemekkel szebbé, jobbá tehetjük a programunkat, de a színek és a grafikus komponensek túlzott használatával sokat ronthatunk, ha nem vigyázunk. A Windows, és az alatta futó alkalmazások felhasználói felületének kialakítása adott szabályokat, elveket követ, amelyet a Microsoft definiált a Windows Style Guidesban. Ennek ismerete nélkül is könnyedén megérthetjük ezeket az irányelveket, ha megnézünk néhány Windows alkalmazást.

Kis eltérés a szabályoktól érdekessé teheti a programot, de ha az egyéniséget túlzásba visszük, bizony lehet, hogy a használhatóságot rontja. Ha egy program másként viselkedik, mint a többi, a felhasználó azt hiheti, hogy nem működik rendesen. Természetesen írhatunk felhasználói dokumentációt, amelyben kitérünk a különbségekre is, de hány felhasználó fogja elolvasni azt a program elindítása előtt? Egy fontos szempont, hogy az általánosan elfogadott felhasználói felület állandóan változik. Minden alkalommal, mikor új program jelenik meg, vele új lehetőségek is, és ezek gyakran megváltoztatják a szabványt.

Nézzük meg ezt a példát: néhány évvel ezelőtt nem voltak még eszköztárak. Megjelent néhány eszköztár nagy, grafikát, és szöveget tartalmazó gombokkal. A következő szabvány kisebb gombokkal, esetlegesen több soros eszköztár volt. Néhány hónappal később, a felhasználók megváltoztathatták a gombokat. Később, egy alkalmazás eszköz tippek nélkül (a sárga hátterű tippek, melyek akkor jelentek meg, ha a kurzor elidőzik egy gomb fölött) már réginek számított. Azután jöttek a mozgatható eszköztárak (amelyeket megfogva átmozgathatunk más helyre a képernyőn), több szín használata (eredetileg tiltott volt), és az új Internet Explorer-szerű eszköztár gombok, amelyek a Delphi 3-ban is megtalálhatók.

Bitmap a gombon

A legegyszerűbb, és legáltalánosabb módja a felhasználói felület fejlesztésének a gombok színesebbé, és ugyanakkor kifejezőbbé tétele. Évekkel ezelőtt a Borland elkezdett képeket tenni a gombokra, hogy kifejezőbbé tegye a őket. Például, az **OK** gombokon egy zöld pipát láthatunk, a **Cancel** gombokon pedig egy piros keresztet. Delphiben kétféle grafikus gomb van:

- A BitBtn (bitmap button) komponens általában tartalmaz szöveget, és grafikát, ahogy a régebbi Borland gombok. A BitBtn gombok hasonlóan viselkednek a Button komponenshez, azaz implementálják a nyomógombokat. Ezeket a gombokat általában formon, vagy párbeszédablakban használjuk.

- A SpeedButton komponensen egyaránt lehet grafika és szöveg, de általában csak egy kis ábrát használunk. Ahogy a neve is utal rá, ezeket általában az eszköztáron használjuk (vagy SpeedBar-on a Borland elnevezése szerint). Működésüket tekintve utánozhatnak nyomógombokat, jelölőnégyzeteket be-, és kikapcsolt állapotokkal, és akár rádiógombokat is. Különböző ábrákkal jelölhetjük a különböző állapotokat.

A fő különbség a BitBtn és a SpeedButton komponens között, hogy az első egy Windows gombon alapul, az owner-drawn technikát (belsőleg) használja a kirajzolásához, míg a második az egy grafikus komponens, nem egy ablakon alapul (nincs ablak leírója). Ezért a SpeedButtonok használata sokkal korlátozottabb, amely különösen fontos, ha sok gombra van szükségünk. A másik dolog amire figyeljünk, hogy a SpeedButtonokra nem fókuszálhatunk (nem választhatjuk ki őket a TAB billentyűvel), és ezért általános formokon és párbeszédablakokon kerüljük a használatukat. Ebben a részben a bitmap gombokról lesz szó, a SpeedButtonokat a következő fejezetre hagyjuk, amely teljes egészében az eszköztárakkal és a státuszsorral foglalkozik.

Ha létrehozunk egy objektumot a BitBtn komponenssel, akkor a Kind tulajdonságot használva beállíthatjuk a hozzá tartozó szöveget, ábrát, és visszatérési értékét. A TBitBtnKind felsorolás választható értékei:

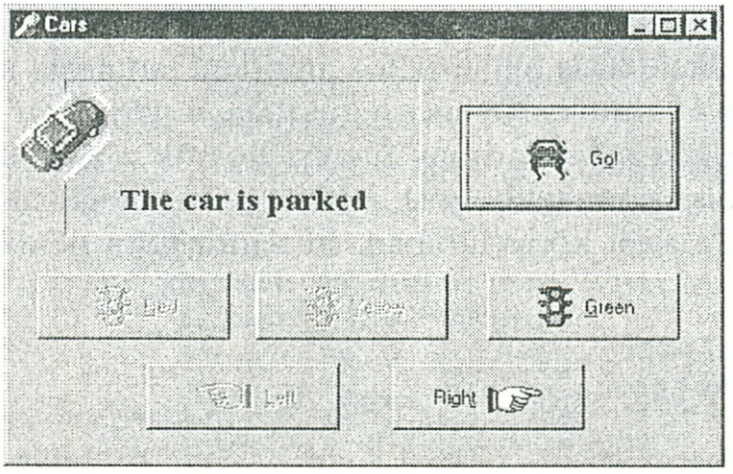
```
type
  TBitBtnKind = (bkCustom, bkOK, bkCancel,
    bkHelp, bkYes, bkNo, bkClose, bkAbort,
    bkRetry, bkIgnore, bkAll);
```

Ha nem egy általános gombot választunk, akkor meg kell adnunk a gomb Caption és Glyph (ikon) tulajdonságát, különben az alapértelmezés szerinti szöveg és kép jelenik meg. Más tulajdonságokkal beállíthatjuk a szöveg és grafika helyét (Layout) és a köztük lévő távolságokat (Spacing).

A másik fontos dolog az átlátszó szín megadása. Amikor készítünk egy képet egy gombra, vagy valamilyen grafikus komponensre, akkor a bal alsó sarokban található pixelt tekinti az átlátszó színnek. Ezért sok alkalmazás, és a Delphi maga is sötét-sárgát használ háttér színnek (a clOlive szín), mert ezt a színt ritkán használják képekben. Ez az érték mutatja meg, hogy a grafika melyik színét vegye átlátszónak, és cserélje ki a gomb háttérszínére. Ha ezt a megközelítést használjuk, akkor rendszer használhatja az alapértelmezés szerinti színeket (világos szürke, sötét szürke, stb.) a gomb különböző állapotaira a képünkkel együtt.

Egy autó a gombon

A grafikus gombok használatának gyakorlására egy egyszerű, és majdnem haszontalan példát készítünk, melynek a Cars (autók) a neve, amelyben több különböző funkciójú bitmap gomb lesz. A programban egy kétállású gomb lesz, amely változtatja a szövegét, és a képét, minden alkalommal, ha megnyomjuk. Lesz még egy három gombból álló csoport, amelyek közül mindig csak egyben aktív. Az utolsó két gomb egyszerűbb. Az egyiken a szöveg és a kép fordítva lesz, hogy szimmetrikus legyen a másikkal, ahogy azt a 11.1. ábrán is láthatjuk.



11.1. ábra *A Cars példa kimenete. A kép a kereten kívülre mozgatható*

A gombok mellett a formon találhatunk egy címkét, egy kép komponenst, és egy Bevel (keret) komponenst. Még sohasem használtuk ezt a két komponenst, ezért röviden leírjuk őket (és utána visszatérünk a példához):

- A kép (Image) komponenst a formok hátthereként szolgáló képek megjelenítésére használhatjuk, általában más komponensek mögött is. Egy bitmap fájlból való beolvasására, vagy fájlba mentésére a komponens Picture (kép) tulajdonságának LoadFromFile, és SaveToFile metódusait kell meghívnunk. Megadhatjuk, hogy a képet eredeti méretben jelenítse meg, vagy a komponens méretéhez igazítsa. Mivel a kép komponens csak a kattintás eseményt támogatja, statikus elemeknek tekinthetjük őket. Később a fejezetben használni fogjuk ezt a komponenst egy képnézegető program létrehozásához.
- A Bevel komponenssel térbeli hatást érhetünk el, amivel a form kontrolljait kiemelhetjük, vagy csoportosíthatjuk. Tartalmazhat dobozokat, kereteket, vagy vonalakat (lásd a Shape tulajdonságát), amelyek lehetnek a síkból kiemeltek, vagy besüllyesztettek, amit a Style tulajdonsággal állíthatunk be. A Bevelnek nincsenek eseményei, csak a felhasználói felület formázására szolgál. A következő fejezetben részletesen is láthatjuk majd, hogy van még egy ilyen komponens, a Panel, amely hasonló a Bevelhez, de sokkal több funkciója van. A TPanel extra képességei erőforrásigényesek, tehát ne használjunk panelt, ha a Bevel is elég.

A Cars példa formján van néhány gomb, és egy keret a bal felső részben, amely egy képet és egy címkét tartalmaz. Ha egyszer egy komponens egy tulajdonságához választunk egy bitmap-et, akkor a bitmap bemásolódik a szöveges leírásba hexadecimális számsorozat formájában. Itt a példa:

```
object CarImage: TImage
  Picture.Data = {
    07544269746D617076020000424D760200000000000076
    0000200000000100040000000000000200000000000000
```

És így tovább még vagy 20 soron keresztül. Ez azt jelenti, hogy nincs szükség a bitmapre a fordításhoz, vagy az alkalmazás futtatásához, mindent tartalmaz a DFM fájl. A későbbiekben azonban a formok szöveges leírásában, amelyek tartalmaznak bitmapet, hivatkozni fogok a fájlra, hogy újraépíthessük a példát (hacsak nem akarunk hexadecimális számokat beírni). A bitmapeket megtalálhatjuk a mellékelt CD-n , mindet a saját könyvtárában. Itt van a Cars példa formjának szöveges leírása:

```
object CarsForm: TCarsForm
  ActiveControl = CarButton
  Caption = 'Cars'
  OnCreate = FormCreate
  OnDestroy = FormDestroy
```

```
  object CarLabel: TLabel
    Alignment = taCenter
    AutoSize = False
    Caption = 'The car is parked'
    Font.Height = -19
    Font.Name = 'Times New Roman'
    Font.Style = [fsBold]
  end
  object Bevel1: TBevel...
  object CarImage: TImage
    Picture.Data = {Cars.bmp}
    Stretch = True
  end
  object CarButton: TBitBtn
    Caption = 'G&o!'
    OnClick = CarButtonClick
    Glyph.Data = {Cars2.bmp}
    Spacing = 12
  end
  object GreenButton: TBitBtn
    Caption = '&Green'
    Enabled = False
    OnClick = GreenButtonClick
    Glyph.Data = {LightG.bmp}
  end
  object RedButton: TBitBtn
    Caption = '&Red'
    TabOrder = 3
    OnClick = RedButtonClick
    Glyph.Data = { LightR.bmp}
  end
  object YellowButton: TBitBtn
    Caption = '&Yellow'
    Enabled = False
    OnClick = YellowButtonClick
    Glyph.Data = {LightY.bmp}
  end
  object LeftButton: TBitBtn
    Caption = '&Left'
    TabOrder = 4
    OnClick = LeftButtonClick
    Glyph.Data = {Left.bmp}
  end
  object RightButton: TBitBtn
    Caption = 'Righ&t'
    OnClick = RightButtonClick
    Glyph.Data = {Right.bmp}
    Layout = blGlyphRight
  end
end
```

Amikor a Left vagy a Right gombot megnyomjuk, akkor a kép balra vagy jobbra mozdul, egészen addig, amíg neki nem ütközik a form szélének (vagyis csak majdnem a szélének, mert egy pár pixel kimarad):

```
procedure TCarsForm.LeftButtonClick(Sender: TObject);
begin
  {move the car, but not outside the area of the form}
  RightButton.Enabled := True;
  CarImage.Left := CarImage.Left - 10;
  {when the left border of the image is less than ten
    pixels from the left border of the form, which is
    at 0}
  if CarImage.Left < 10 then
  begin
    RightButton.SetFocus;
    LeftButton.Enabled := False;
  end;
end;
```

Ahogy azt a 11.1. ábrán is láthatjuk, ez a megoldás messze nem a legjobb, mert a Bevel szélénél kellene lennie a határnak. Ez a kis változtatás az olvasóra marad gyakorlásképpen.

Amikor a három közlekedési lámpa bitmap-es gomb közül valamelyiket megnyomjuk, akkor a kód kikapcsolja, és egy másikat kapcsol be helyette, körbejárásos megközelítést alkalmazva: a piros gomb után a zöld gomb következik, a zöld gomb után a sárga gomb, a sárga gomb után a piros. Itt van egy a három metódus közül:

```
procedure TCarsForm.RedButtonClick(Sender: TObject);
begin
  {after the red, green}
  GreenButton.Enabled := True;
  GreenButton.SetFocus;
  RedButton.Enabled := False;
end;
```

A RedButton kikapcsolása előtt meghívjuk a GreenButton objektum SetFocus metódusát. Amikor kikapcsoljuk a gombot, akkor a fókusz (majdnem) véletlenszerűen átkerül más objektumra. Ha ezután hívjuk meg a SetFocust, akkor a fókusz zavaró villanással áttevődik az adott objektumra. Ha a gomb kikapcsolása előtt hívjuk meg a SetFocust, akkor azonnal az általunk meghatározott gombra fókuszál. Ennek az előnye az, hogy ha a szóköz billentyűvel „kattintunk" a gombra, akkor láthatjuk, ahogy a fókusz gombról gombra kerül, meghatározott sorrendben.

Ugyanez történik a Left, és Right gombokkal is. Ha folyamatosan nyomjuk a szóköz billentyűt, akkor a kép nekiütközik a keretnek, és visszafelé indul úgy,

hogy a fókuszt a másik gomb kapja meg, amely az ellenkező irányba mozgatja a képet. Billentyűparancsokat is használhatunk (ezeket a gombok címkéjében egy & jellel definiálhatjuk). Ha futási idő alatt változtatjuk a gombok címkéjét (lásd az alábbi kódot), akkor ne felejtsük el megfelelően beállítani az & jelet, hogy a billentyűparancsok megfelelőek legyenek.

A példa legösszetettebb kódrészlete az autós gomb. A gombra kattintva megváltoztatjuk a képet és a címke szövegét magán a gombon, a Kép (Image) komponens kép (picture) tulajdonságát, és a címke Caption tulajdonságát. A kód a form CarStopped privát logikai változóját is beállítja, hogy nyomon követhessük az aktuális állapotot.

Hogy elkerüljük a képhez és a gombhoz a bitmap-ek folyamatos betöltését, a következő sort használjuk:

```
CarButton.Glyph.LoadFromFile ('cars2.bmp');
```

Hozzáadtunk a formhoz két bitmap objektumot, amelyek tárolják a két grafikus képet. A formnak három mezője van:

```
private
  CarStopped: Boolean;
  Car1Bmp, Car2Bmp: TBitmap;
```

Ezeket a privát mezőket a form létrehozásakor inicializáljuk:

```
procedure TCarsForm.FormCreate(Sender: TObject);
begin
  {set the flag and load the two bitmaps of the car}
  CarStopped := True;
  Car1Bmp := TBitmap.Create;
  Car2Bmp := TBitmap.Create;
  Car1Bmp.LoadFromFile ('cars.bmp');
  Car2Bmp.LoadFromFile ('cars2.bmp')
end;
```

Az inicializálás után már használhatjuk ezt a két bitmap objektumot a kép, és a grafikus gomb objektumok értékének beállítására. Természetesen két különböző eset lehetséges a CarStopped értékétől függően. Van egy speciális eset is. Az autó nem mozoghat, ha a piros lámpa aktív – vagyis a RedButton be van kapcsolva. Ekkor egy figyelmeztető szöveg jelenik meg, és semmi nem történik. A CarButton click metódusának első részét itt láthatjuk:

```
procedure TCarsForm.CarButtonClick(Sender: TObject);
begin
  {a car should not start if the light is red}
  if CarStopped then
    if RedButton.Enabled then
      MessageDlg ('No turn on red, please!',
        mtWarning, [mbOK], 0)
```

```
        else {if was stopped and it is not red}
        begin
          {change the bitmaps and captions}
          CarButton.Glyph := Car1Bmp;
          CarButton.Caption := 'St&op';
          CarImage.Picture.Graphic := Car2Bmp;
          CarLabel.Caption := 'The car is on the road';
          CarStopped := False;
        end
      else {if the car was moving, regardless of the lights}
        ...
```

A kód második része ugyanezt csinálja, egy másik bitmapet és szöveget használva. a program működéséhez a két autó képnek az program könyvtárában kell lennie, mivel ezeket futás közben töltjük be. Ne felejtsük el felszabadítani a két bitmap objektumot a form OnDestroy eseményében.

Változó kép a gombon

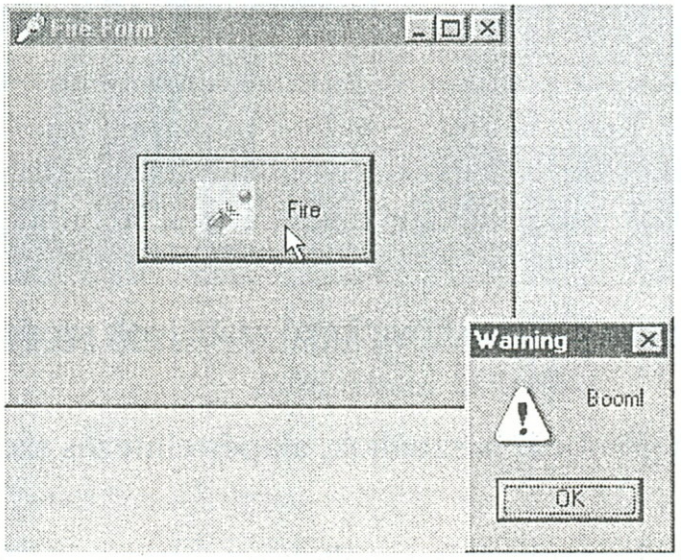
Ahogy az előző példában láthattuk, a bitmap gombokat könnyű használni, és szebbé tehetik a programot, mint ha csak általános nyomógombokat használnánk (Button komponenst). Az animált gombbal még tovább tudjuk fejleszteni a külső megjelenést. Alapvetően két féle animált gomb van – gombok, amelyek kissé változtatják a képüket ha megnyomjuk őket, és gombok, amelyek mozgó képet tartalmaznak az aktuális állapotuktól függetlenül. Mindkét fajtára mutatok egy egyszerű példát (Fire és World), és egy-egy bonyolultabbat is az érdekesség kedvéért.

A kétállású gomb

Az első példa, a Fire, egy egyszerű formból áll, amelyen egy bitmap gombot tartalmaz. A gomb címkéje a „Fire", amely kapcsolódik egy ágyút megjelenítő képhez. Képzeljük el ezt a gombot egy játékprogram részeként. Ha a gombot megnyomjuk, akkor a kép egy tüzelő ágyú képére változik, ahogy azt a 11.2. ábrán is láthatjuk. Ahogy felengedjük a gombot, akkor a kiinduló kép töltődik be újra. Közben a program megjelenít egy üzenetet, hogy a felhasználó valóban rákattintott a gombra (ha a felhasználó megnyomja a gombot és közben elmozgatja az egeret és ott engedi fel, akkor az OnClick esemény nem fut le, habár a bitmap átmenetileg megváltozik).

Hogy ezt a programot meg tudjuk írni, a gomb három eseményét kell kezelnünk: OnMouseDown, OnMouseUp, és az OnClick. A három metódus kódja rendkívül egyszerű:

```
procedure TForm1.BitBtnFireMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y:
Integer);
begin
  {load firing cannon bitmap}
  if Button = mbLeft then
    BitBtnFire.Glyph.LoadFromFile ('fire2.bmp');
end;

procedure TForm1.BitBtnFireMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y:
Integer);
begin
  {load default cannon bitmap}
  if Button = mbLeft then
    BitBtnFire.Glyph.LoadFromFile ('fire.bmp');
end;

procedure TForm1.BitBtnFireClick(Sender: TObject);
begin
  MessageDlg ('Boom!', mtWarning, [mbOK], 0);
end;
```



11.2. ábra *A kép amelyet a Fire példa jelenít meg a gomb megnyomásakor, és a „Boom" üzenet*

Amikor nyomva tartjuk a bal egér gombot a bitmap gombon, akkor az besüllyed. Ha ezután elmozgatjuk az egeret a gombról az egérgombot nyomva tartva, akkor a bitmap gomb felenged, de nem futtat le OnMouseUp eseményt, tehát a tüzelő ágyú képe ott marad. Ha később elengedjük a bal egérgombot a bitmap gomb felületén kívül, akkor lefut az OnMouseUp esemény.

Windowsban az a szabály, hogy minden egér esemény átadódik az ablaknak, vagy a kurzor alatti komponensnek. Hogy küldhetünk egy egér eseményt a gombnak, ha a felületén kívül engedjük fel az egér gombot? Ez a viselkedés az ún. *mouse capture* technikán múlik. Egy ablak dönthet úgy, hogy elfog minden további egér eseményt függetlenül a kurzor alatti ablaktól. Pontosítva a szabályt, tehát minden egér esemény a kurzor alatti ablakhoz vagy ahhoz a komponenshez kerül, amelyik „elfogta" az egér inputot.

Tipp

Elfoghatjuk az egérműveleteket a Windows API SetCapture funkciójával, és később befejezhetjük ezt a műveletet a ReleaseCapture funkcióval. Az egér eseményeinek elfogása nagyon érdekes lehet néhány alkalmazásban, és ez az alapja minden műveletnek, ahol egérrel húznunk, ahogy azt a későbbi fejezetekben is látni fogjuk (különösen a 14. fejezetben).

Több kép egy bitmapben

A Fire példa manuális megközelítést használt. Betöltöttünk két bitmapet és a Glyph tulajdonság értékével változtattuk a képeket. A BitBtn komponens tud több képet is automatikusan kezelni. Készíthetünk egyetlen bitmapet, amely több képet tartalmaz, és ezt az értéket beállíthatjuk a NumGlyphs tulajdonságban. Az összes részkép méretének meg kell egyeznie, mivel a bitmap egyenlő részekre van osztva.

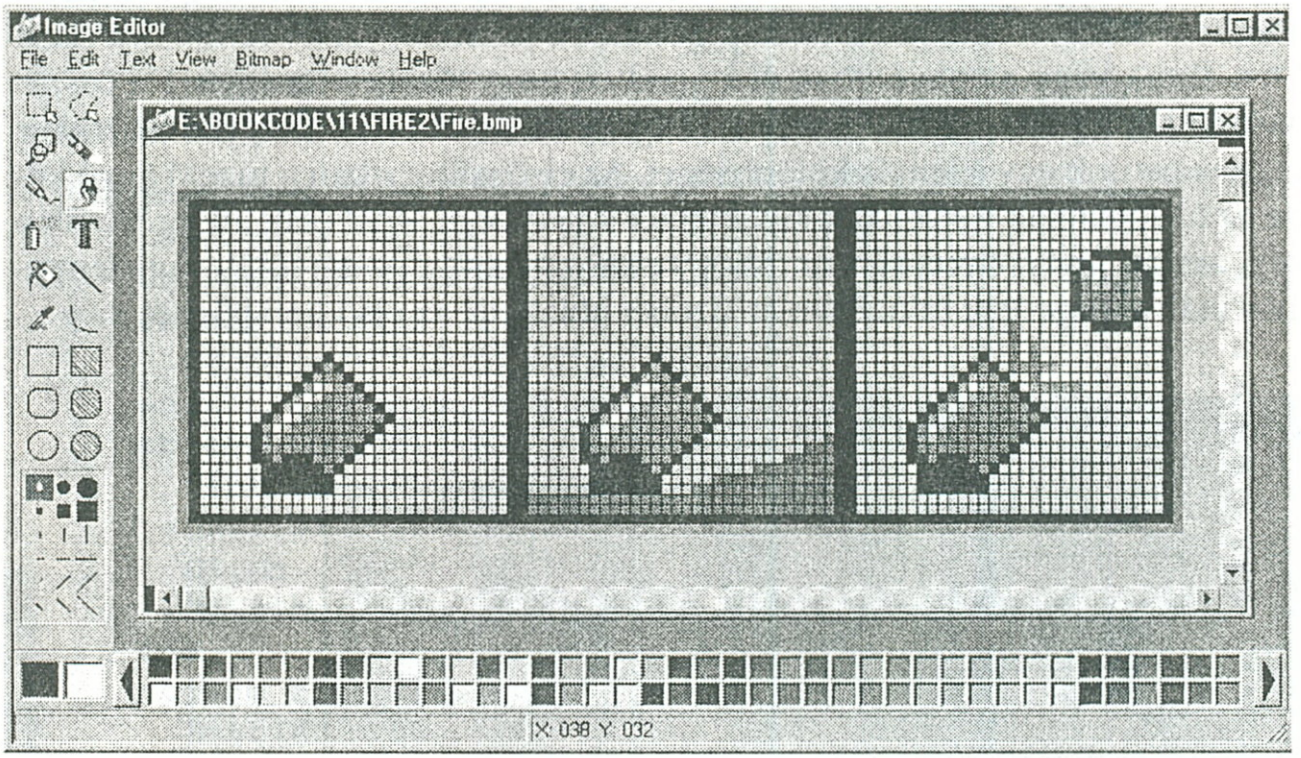
Ha egynél több képet tartalmaz a bitmap, akkor a következő szabályok szerint használhatók:

- Az első bitmapet a felengedett gombnál használjuk, alapértelmezés szerinti állapot.
- A második bitmapet a kikapcsolt gombnál használjuk.
- A harmadik bitmapet a benyomott gombnál használjuk.
- A negyedik bitmapet a benyomva maradó gombnál használjuk, amikor a gombok jelölőnégyzetként működnek.

Általában csak egy képet adunk meg, és a többi ebből automatikusan létrehozható egyszerű grafikai változtatásokkal. Egyszerű megadni egy második, harmadik, és negyedik képet is. Ha nem adjuk meg mind a négy képet, akkor a hiányzó képek az elsőből kiindulva lesznek automatikusan kiszámolva. Az utolsó kettő kép hiányozhat csak. Nem adhatunk meg egy normál, és egy

benyomott képet a közöttük lévő kikapcsolt állapotot megjelenítő bitmap nélkül.

A példánk új változatában a Fire2-ben csak az első és a harmadik képre van szükségünk, de rá vagyunk kényszerítve a második kép megadására is. Hogy lássuk ennek a második képnek a működését, hozzáadtunk egy jelölőnégyzetet a gomb kikapcsolására. Az új verzióhoz készítettünk egy 32×96 pixel felbontású bitmapet (lásd 11.3. ábra), amit a bitmap Glyph tulajdonságánál használunk. Meglepetésünkre a NumGlyphs tulajdonság automatikusan 3-ra lett állítva, mivel a bitmap háromszor olyan széles, mint magas.



11.3. ábra *A bitmap a Fire2 példa három képével, ahogy a Delphi Image Editorában láthatjuk*

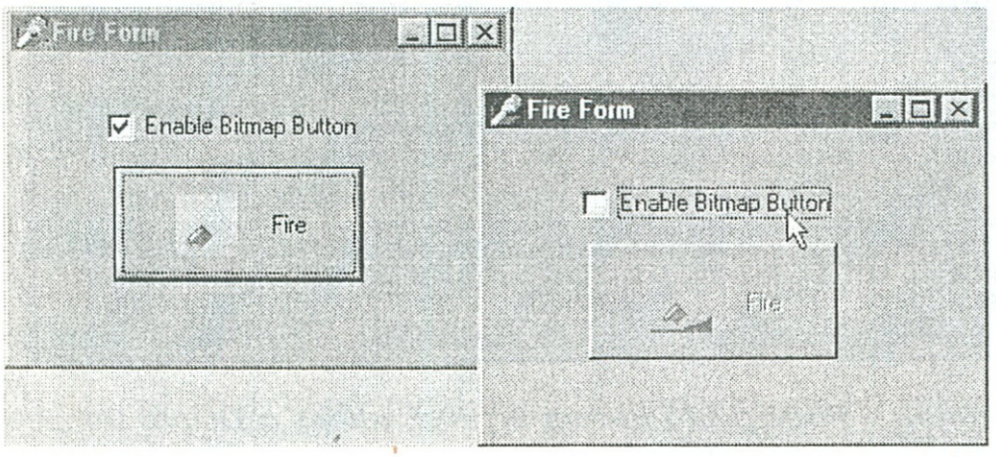
Kiszedtük a BitBtnFireMouseDown és BitBtnFireMouseUp metódusokat, amelyekre most már nincs szükség. A következő lépésben egy jelölőnégyzetet adtunk a formhoz, amellyel ki-, és bekapcsolhatjuk a gombot (így láthatjuk a kikapcsolt állapotnak megfelelő ábrát), és az OnClick eseményéhez írtunk egy sort:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  BitBtnFire.Enabled := CheckBox1.Checked;
end;
```

Hozzáadtunk még hang megszólaltatási lehetőséget is, amely egy WAV fájlt játszik le a gomb megnyomásakor (az MMSystem egység PlaySound funkciójának meghívásával). A kód nagyon egyszerű, ezzel is érzékeltetni szeretném, milyen egyszerű ilyen jellegű részeket készíteni (E témát járjuk majd körbe a 31. fejezetben):

```
procedure TForm1.BitBtnFireClick(Sender: TObject);
begin
  PlaySound ('Boom.wav', 0, snd_Async);
  MessageDlg ('Boom!', mtWarning, [mbOK], 0);
end;
```

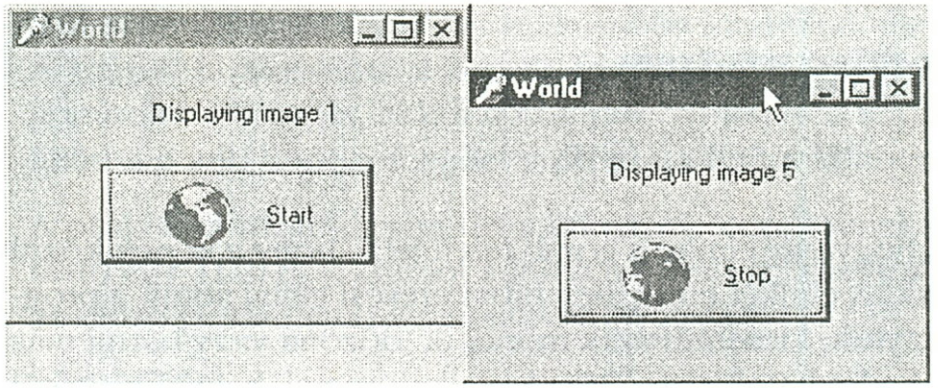
Ha futtatjuk a programot, akkor kétféleképpen változtathatjuk meg a képet a gombon. Kikapcsolhatjuk a bitmap gombot a jelölőnégyzetet használva (lásd 11.4. ábra), vagy megnyomhatjuk a gombot, hogy tüzeljen az ágyú. Ha összehasonlítjuk a program kimenetét a 11.2. ábrán látható tüzelő ágyúéval, akkor észrevehetünk egy jelentős különbséget. Az első változatban a tüzelő ágyú képe az üzenetablak bezárásáig rajtamaradt a gombon. Most csak addig látszik a kép, amíg a gomb nyomva van. Ahogy elmozdulunk a gomb fölül, vagy lenyomás után elengedjük (aktiválva az üzenet ablakot), az első kép jelenik meg.



11.4. ábra *A Fire2 példa engedélyezett és letiltott bitmap gombjai, az alkalmazás két külön másolatában*

A forgó földgömb

Az animáció második példájában, a Worldben egy gombon látszik a Föld, amely lassan forog, megmutatva a különböző kontinenseket. Láthatunk néhány példát a 11.5. ábrán, de természetesen futtassuk a programot, hogy láthassuk a kimenetét. Az előző példában a kép akkor változott, ha megnyomtuk a gombot. Most a kép magától változik automatikusan. Ez a Timer (Időzítő) komponensnek köszönhető, amely adott időintervallumonként üzenetet kap. Az időzítőkről és a velük kapcsolatos témákról a 25. fejezetben lesz szó.



11.5. ábra *A futó World program néhány példája*

A példában található komponensek tulajdonságainak összegzését itt láthatjuk:

```
object WorldForm: TWorldForm
  Caption = 'World'
  OnCreate = FormCreate
  object Label1: TLabel
    Caption = 'Displaying image 1'
  end
  object WorldButton: TBitBtn
    Caption = '&Start'
    OnClick = WorldButtonClick
    Glyph.Data = {W1.bmp}
    Spacing = 15
  end
  object Timer1: TTimer
    Enabled = False
    Interval = 500
    OnTimer = Timer1Timer
  end
end
```

A Timer komponens elindul és megáll (bekapcsol és kikapcsol), amikor a felhasználó megnyomja a világot ábrázoló bitmap gombot. A gombon van a „Start" címke, amely „Stop"-ra változik, ha az időzítő aktív. Tehát a gombot megnyomva elindíthatjuk, és megállíthatjuk az animációt:

```
procedure TWorldForm.WorldButtonClick(Sender: TObject);
begin
  if Timer1.Enabled then
  begin
    Timer1.Enabled := False;
    WorldButton.Caption := '&Start';
  end
  else
```

```
    begin
      Timer1.Enabled := True;
      WorldButton.Caption := '&Stop';
    end;
  end;
```

A 11.5. ábrán láthatjuk, hogy egy címke a gomb felett jelzi, hogy a képek közül éppen melyiket láthatjuk. A címkét eredetileg hibakeresési céllal adtuk a programhoz, de végül otthagytuk; kicserélhetjük mondjuk időzóna vagy bármi más információ kijelzésére, ha kedvünk tartja. Minden alkalommal, ha az időzítőtől üzenet érkezik, a kép és a címke megváltozik:

```
procedure TWorldForm.Timer1Timer(Sender: TObject);
begin
  Count := (Count mod 16) + 1;
  Label1.Caption := 'Displaying image ' +
    IntToStr (Count);
  WorldButton.Glyph.LoadFromFile (
    'w' + IntToStr (Count) + '.bmp');
end;
```

Ebben a példában a Count a form egyik mezője, amelyet a FormCreate metódusban inicializálunk 1-re. Minden időzítő intervallumban Countot növeljük a 16-tal osztott maradékával és karakterlánccá alakítjuk („w"-vel az elején).

Megjegyzés

A modulus művelet az egészek osztásából a maradékot adja vissza. Ez azt jelenti, hogy a `Count mod 16` mindig 0-15 közötti értéket ad vissza. Egyet adva ehhez a visszatérési értékhez megkapjuk a bitmap sorszámát, amely a 1-16 intervallumba esik.

Ennek a korlátozásnak egyszerű a magyarázata: 16 darab Föld bitmapünk van. A bitmap fájlok nevei W1.BMP, W2.BMP, és így tovább, így a programból egyszerűen el tudjuk érni őket, futási idő alatt létrehozva a neveket.

Bitmap lista, erőforrások használata, és a rajztábla

A World program működik, de nagyon lassú, néhány okból kifolyólag. Először is minden időzítő intervallumban be kell olvasnia egy fájlt a lemezről; egy lemezgyorsító tár (disk cache) felgyorsíthatja, de ez nem a legjobb megoldás. A fájlok lemezről való beolvasása mellett a programnak létre kell hoznia, és meg

kell semmisítenie Windows bitmap objektumokat, és ez is időt vesz igénybe. A második probléma a képek frissítésétől függ: Amikor megváltoztatjuk a gombon a bitmapet, akkor a komponens teljesen letörlődik, és újrarajzolódik. Ezért a gomb képe remeghet a program futása közben.

A probléma megoldására (és a bitmapek kezelésének egy másik megközelítésére) egy TList objektumot adtunk a formhoz, amely a bitmapek listáját tárolja, az előző példa újabb verziójaként (World2):

```
type
  TWorldForm = class(TForm)
    ...
  private
    Count: Integer;
    BitmapsList: TList;
  end;
```

Az összes bittérkép betöltődik a program indításakor, és megsemmisül kilépéskor. Minden időzítő intervallumban a program megjelenít egyet a bitmap listából a bitmap gombon. A listával elkerülhetjük a bittérképek állandó töltögetését, de az összes fájlra szükség van a futtatható fájl könyvtárában. Erre a problémára a megoldás az, hogy a különálló bitmap fájlokat átmozgatjuk az program erőforrás (resource) fájljába. Ezt egyszerűbb megcsinálni, mint elmagyarázni.

Az erőforrások grafikus vagy szöveges adatok, amelyek egy Windows programhoz kapcsolódnak. A legfontosabb erőforrások Delphiben a bittérképek, ikonok, fontok, string táblák, és a formok képei (az alkalmazás erőforrásainak semmi köze a rendszer szabad erőforrásaihoz – a kifejezés kétféle használata teljesen különböző dolgokat takar). Az erőforrásokról részletesen beszélünk a 27. fejezetben, de grafikus példákban gyakran használjuk őket addig is (később ebben a fejezetben ismét előkerülnek).

Tipp

Alapvetően három ok van, amiért érdemes erőforrásokat használni Windows alkalmazásokban. Az első az, hogy némelyik grafikus elemekre hivatkozik, és ehhez megfelelő szerkesztő szükséges; a második sokkal fontosabb ok, mert az erőforrások a memóriában vannak különleges módon vannak kezelve (lásd a 26. fejezetet a Windows memória kezeléséről bővebben); a harmadik, hogy a Windows alkalmazások más nyelvű változatainak elkészítéséhez ne kelljen a forráskódot megváltoztatni.

Ha forrásfájlokat használunk bitmap fájlok helyett, először a bitmap állományokat be kell másolnunk az erőforrás fájlba. Használhatjuk a Delphi Image Editor programját is egy új erőforrás-fájl létrehozására, ahol egyenként megnyithatjuk a bitmap fájlokat és bemásolhatjuk onnan a fájlba, habár ez eléggé lassú és monoton munka.

Figyelem!

Ne adjuk az új bitmapeket a program WORLD.RES fájljához. Ezt a fájlt a Delphi bizonyos körülmények között frissíti, törölve az aktuális tartalmat!

A jobbik megoldás az, hogy írunk egy forrás scriptet (egy .RC fájlt) amely a bitmap fájlok és más szükséges fájlok listáját tartalmazza. A **File** menü **New** parancsát kiválasztva megnyitjuk az Object Repositoryt, ott kattintsunk a **New** oldalon a *Text* ikonra. A Delphi által létrehozott üres szövegmezőbe írjuk be a következő kódot:

```
W1   BITMAP   "W1.BMP"
W2   BITMAP   "W2.BMP"
W3   BITMAP   "W3.BMP"
... és így tovább
```

Ha elkészültünk az RC fájllal (WORLDBMP.RC nevet adtuk neki), akkor lefordíthatjuk a Delphi erőforrás fordítójával (resource compiler) egy RES fájlba (a fordító a BRCC32 parancssoros alkalmazás, amelyet a Delphi BIN könyvtárában találhatunk meg):

```
BRCC32 WORLDBMP.RC
```

Utána adjunk a program forráskódjához egy fordítási direktívát, hogy hozzácsatolja az új erőforrás fájlt. Ezt a formot leíró egységben rakhatjuk, a form erőforrás fájljának (DFM) csatoló direktívája után:

```
{$R *.DFM}
{$R WORLDBMP.RES}
```

Ha definiáltuk az alkalmazás erőforrásait, akkor be kell töltenünk a bitmap fájlokat onnan. A Delphi komponensei nem támogatják ezt közvetlenül, így a Windows API LoadBitmap függvényét kell használnunk. E függvénynek két paraméterre van szüksége: az alkalmazás leírója (HInstance), amely egy globális változó a Delphiben; és egy stringre, ami az erőforrás nevét tartalmazza.

Új!

A Delphi 3 definiál egy új globális változót a MainInstance-t, ami utal a fő futtatható fájl HInstance-jára. A legtöbb esetben használhatjuk az egyiket és a másikat felcserélve.

Természetesen a LoadBitmap API funkciónak egy PChar stringet kell átadni, nem egy Pascal stringet, tehát szükség lehet adatkonverzióra is. A FormCreate metódus kódja:

```
procedure TWorldForm.FormCreate(Sender: TObject);
var
  I: Integer;
  Bmp: TBitmap;
  BmpName: string;
begin
  Count := 1;

  {load the bitmaps from the resources of the
  application to the list of TBitmap objects}
  BitmapsList := TList.Create;
  for I := 1 to 16 do
  begin
    Bmp := TBitmap.Create;
    BmpName := 'W' + IntToStr (I);
    {load the bitmap from the resources:
    LoadBitmap is a Windows API function}
    Bmp.Handle := LoadBitmap (HInstance,
        PChar (BmpName));
    BitmapsList.Add (Bmp);
  end;
end;
```

Megjegyzés

Később a fejezetben látni fogjuk, hogyan használjuk a Delphi ImageList komponensét, amely a Windows 95 egyik általános kontrolján alapul. Ez a példa létrehoz a semmiből egy egyszerű objektumlistát, amelyet BitmapsListnek hívnak, hogy megmutassa miből áll egy képlista.

Egy megoldandó probléma van még. Hogy tudjuk megoldani a World képeinek gördülékeny váltását egyikről a másikra? Megpróbáltuk megtenni ezt a TBitBtn osztállyal, de több akadálya is volt. Egy kis idő múlva azonban kitaláltunk egy teljesen más megoldást. Miért ne rajzolnánk a bitmapet egy Canvas komponensre a Draw metódust használva? Sajnos, a bitmap gomb felszíne közvetlenül ne érhető el, ezért egy új komponenst adtunk a formhoz, a PaintBox-ot. A dolgok így sem működtek. Nem tehetünk egy rajzdoboz komponenst egy gombra, csak a formra. Még ha egy gomb fölé helyezzük, a Delphi a rajzdobozt akkor is a formhoz kapcsolja, azaz a gomb mögé (alá) helyezi. A rajzdoboz komponens valójában nem ablakon alapul, hanem a szülő kontrolját használja, jelen esetben a formot. A megoldás? Nagyon könnyű, ha egyszer végiggondoltuk – változtassuk meg a szülőt futási idő alatt, ezt a kódot írva a megfelelő helyre (a FormCreate metódusban):

```
PaintBox1.Parent := WorldButton;
PaintBox1.SetBounds (
  WorldButton.Margin,
  (WorldButton.Height - Bmp.Height) div 2,
  Bmp.Width + 2,
  Bmp.Height + 2);
```

A rajzdoboz vízszintes pozíciója a bitmap gomb ikonjának a margójától (Margin) függ. A rajzdoboz méretet akkora, mint a bitmap, és függőlegesen középre van igazítva a gombon belül. Ha megfelelően beállítottuk a rajzdoboz komponenst, akkor felrajzolhatjuk a Timer1Timer metódusban:

```
procedure TWorldForm.Timer1Timer(Sender: TObject);
begin
  Count := (Count mod 16) + 1;
  Label1.Caption := Format ('Displaying image %d',
    [Count]);

  {draw the current bitmap in the canvas placed
  over the bitmap button}
  PaintBox1.Canvas.Draw (1, 1,
    BitmapsList.Items[Count-1]);
end;
```

Ez a kód másolja fel a rajzdoboz komponens felületére a bitmapet – ami a szülőjének a felszíne –, azaz a bitmap gombra. Egy másik fontos tennivaló egy megfelelő FormDestroy metódus definiálása, a bitmapek felszabadítására:

```
procedure TWorldForm.FormDestroy(Sender: TObject);
begin
  {free each element and the list itself}
  BitmapsList.Clear;
  BitmapsList.Free;
end;
```

Az utolsó probléma a rajzdoboz mozgatása, amikor a bal egérgombot lenyomjuk, vagy felengedjük fölötte (ami a gomb OnMouseDown és az OnMouseUp eseményeiben van). A rajzdoboz mozgatása mellett frissítenünk kell a bitmap ábrát, mert a Delphi automatikusan kirakja, a gomb újrarajzolása alatt. Ha nem így csináljuk, akkor addig látjuk a kezdőképet, amíg az időzítő intervalluma le nem jár (ami eltarthat egy darabig, ha megállítottuk!).

Itt a két metódus kódja:

```
procedure TWorldForm.WorldButtonMouseDown(
     Sender: TObject;
     Button: TMouseButton;
     Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
     {paint the current image over the button}
     PaintBox1.Left := PaintBox1.Left + 2;
     PaintBox1.Top := PaintBox1.Top + 2;
     WorldButton.Glyph.Assign (
       BitmapsList.Items[Count-1]);
  end;
end;

procedure TWorldForm.WorldButtonMouseUp(
     Sender: TObject;
     Button: TMouseButton;
     Shift: TShiftState;
     X, Y: Integer);
begin
  if Button = mbLeft then
  begin
     PaintBox1.Left := PaintBox1.Left - 2;
     PaintBox1.Top := PaintBox1.Top - 2;
     WorldButton.Glyph.Assign (
       BitmapsList.Items[Count-1]);
  end;
end;
```

Megjegyzés

Ha egy PaintBox komponenst egy gomb elé helyezünk, akkor fellép egy probléma: amikor a felszínére kattintunk, akkor a gomb nem lesz megnyomva. Hogy kibővítsük a programot, a PaintBox1 komponens OnClick eseményét összekapcsoltuk a gomb OnClick eseményét kezelő metódussal. Ez így működik, de ha erre a területre kattintunk, akkor sem fogjuk látni a gombot benyomódni.

Az Animáció kontrol

Van egy jobb mód is az animáció létrehozására, mint képek egymás utáni megjelenítése. Használjuk a Windows 95 Animate kontrolját. A Borland hozzáadott a Delphi 3-hoz egy komponenst, amibe ezt beleágyazta. Az animáció vezérlő az AVI fájlokon (Audio Video Interface) alapul, bitmapek sorozata hasonlóan egy filmhez.

Megjegyzés

Valójában az animáció kontrol csak azokat az AVI fájlokat jelenítheti meg, amelyeknek folyamatos a lejátszása, tömörítetlenek, vagy RLE8 tömörítést használnak, nincsen paletta változás bennük, és ha van bennük hang, akkor azt figyelmen kívül hagyja. Gyakorlatban az ezeknek a követelményeknek megfelelő fájlok bitmapek sorozatából állnak, és nem valódi filmen alapulnak.

Az Animáció vezérlőnek két féle adatforrást használhat:

- Alapulhat AVI fájlokon, a fent leírt módon: elérhetjük a FileName tulajdonság megfelelő beállításával.
- Használhat egy speciális beépített Windows 95 animációt, ami az általános kontrolkönyvtár része: ezt elérhetjük a CommonAVI tulajdonság egy lehetséges értékének kiválasztásával (amely felsoroláson alapul).

Ha a formra teszünk egy animáció vezérlőt, kiválasztunk egy animációt a fent leírt módszerek egyikével, és végül igazra állítjuk az Active tulajdonságát, akkor elindíthatjuk az animációt, akár szerkesztési idő alatt is. Alapértelmezés szerint az animáció folyamatosan fut, újraindulva, ha a végére ért. Szabályozhatjuk ezt a működést a Repetitions (ismétlések) tulajdonsággal. Ha ezt a tulajdonságot -1-re állítjuk, akkor végtelen lesz a lejátszás.

Megadhatjuk a kezdő és a végső képet (frame) a sorozatból a StartFrame és a StopFrame tulajdonságokkal. Ez a három tulajdonság (kezdő pozíció, végső pozíció, és az ismétlések száma) megfelel az általunk gyakran használt Play metódus, három tulajdonságának. Alternatívaként beállíthatjuk a tulajdonságokat, és meghívhatjuk a Start metódust. Futási idő alatt is elérhetjük az összes képet a FrameCount tulajdonságot használva: használhatjuk az animáció elindítására az elejétől a végéig. Végezetül, finomabb beállításra, használhatjuk a Seek metódust, amely megjeleníti a megadott képet.

Az összes metódust használtuk egy egyszerű bemutató (demo) programban, amely használhat fájlokat és a Windows 95 általános animációit. A programban

választhatunk egy fájlt, vagy egyet az animációk közül a listapanelt használva. A listapanelhez olyan sorrendben adtuk hozzá az elemeket, ahogy a TCommonAVI felsorolásban lévő elemek vannak:

```
object ListBox1: TListBox
  Items.Strings = (
    '[Use an AVI file]'
    'Find Folder'
    'Find File'
    'Find Computer'
    'Copy Files'
    'Copy File'
    'Recycle File'
    'Empty Recycle'
    'Delete File')
  OnClick = ListBox1Click
end
```

Ennek a struktúrának köszönhetően, ha a felhasználó a listapanelra kattint, akkor egyszerűen megnézzük a kiválasztott elem sorszámát, ami megfelel a felsorolás adattípusú CommonAVI tulajdonságnak:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
 Animate1.CommonAVI := TCommonAVI (ListBox1.ItemIndex);
 if (ListBox1.ItemIndex = 0) and
    OpenDialog1.Execute then
       Animate1.FileName := OpenDialog1.FileName
end;
```

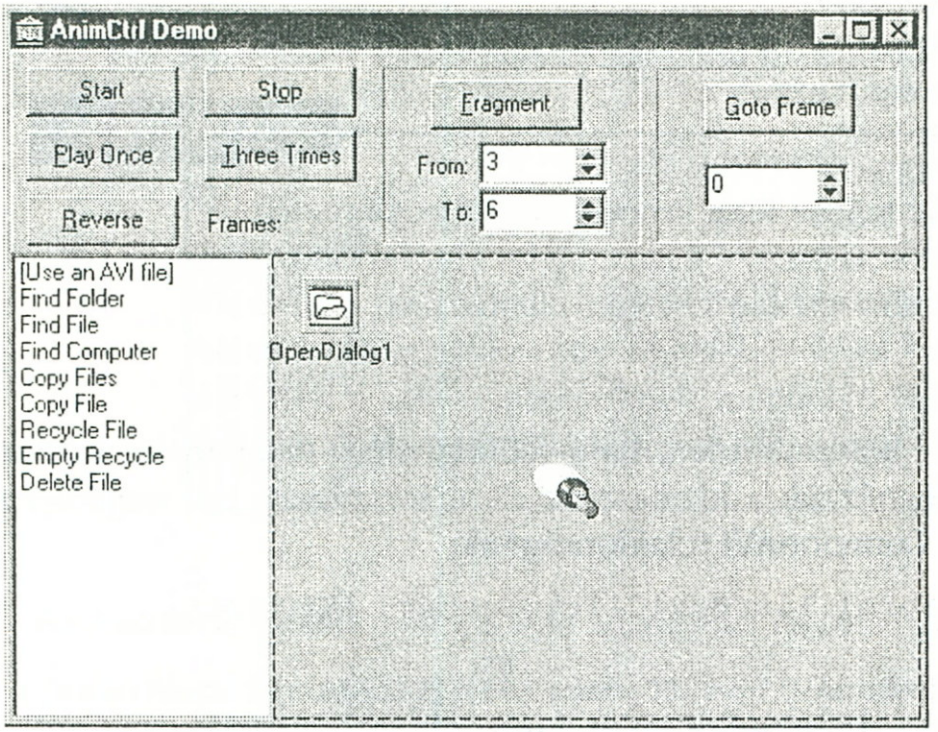
Ahogy láthatjuk, amikor az első elemet választjuk ki (az értéke caNone), akkor a program automatikusan betölt egy AVI fájlt, az OpenDialog (fájl megnyitó) komponenssel. A form legfontosabb komponense az animáció vezérlő. Itt van a szöveges leírása:

```
object Animate1: TAnimate
  AutoSize = False
  Align = alClient
  CommonAVI = caFindFolder
  OnOpen = Animate1Open
end
```

A komponenst a felhasználói felülethez igazítottuk, hogy a felhasználó könnyedén tudja változtatni a méretét az animáció képeinek aktuális méretére. Ahogy láthatjuk, definiáltunk egy kezelőt a komponens OnOpen eseményében:

```
procedure TForm1.Animate1Open(Sender: TObject);
begin
  LblFrames.Caption := 'Frames ' +
    IntToStr (Animate1.FrameCount);
end;
```

Amikor egy új fájlt (vagy egy általános animációt) megnyitunk, akkor a program egyszerűen kijelzi egy címkén a képkockák számát. Ez a címke egy panelen található több gombbal, és néhány léptetőgombbal (SpinEdit control), amely eszköztárként funkcionál. Ezeket láthatjuk a 11.6. ábrán lévő szerkesztés alatt álló formon.



11.6. ábra *Az AnimCtrl példa formja szerkesztési idő alatt. Az animáció kontrol valóban mutatja az animációt, még a program futtatása előtt*

A Start, Stop, és Play Once gomboknak van a legegyszerűbb kódja:

```
procedure TForm1.BtnStartClick(Sender: TObject);
begin
  Animate1.Active := True;
end;

procedure TForm1.BtnStopClick(Sender: TObject);
begin
  Animate1.Stop;
end;

procedure TForm1.BtnOnceClick(Sender: TObject);
begin
  Animate1.Play (0, Animate1.FrameCount, 1);
end;
```

A dolog érdekesebbé válik az animációt háromszor lejátszó, vagy egy részét megjelenítő kóddal. Mindkét eljárás a Play metóduson alapul:

```
procedure TForm1.BtnTriceClick(Sender: TObject);
begin
  Animate1.Play (0, Animate1.FrameCount, 3);
end;

procedure TForm1.BtnFragmentClick(Sender: TObject);
begin
  Animate1.Play (SpinEdit1.Value, SpinEdit2.Value, -1);
end;
```

Az utolsó két gomb eseménykezelője a Seek metóduson alapul. A Goto gomb a SpinEdit komponens által mutatott képkockára pozícionál, amíg a Reverse gomb egyesével visszafelé lépteti az animációt, képkockánként szünetet tartva:

```
procedure TForm1.BtnGotoClick(Sender: TObject);
begin
  Animate1.Seek (SpinEdit3.Value);
end;

procedure TForm1.BtnReverseClick(Sender: TObject);
var
  Init: TDateTime;
  I: Integer;
begin
  for I := Animate1.FrameCount downto 0 do
  begin
    Animate1.Seek (I);
    // wait 30 milliseconds
    Init := Now;
    while Now < Init + EncodeTime (0, 0, 0, 30) do
      Application.ProcessMessages;
  end;
end;
```

Az utolsó metódust továbbfejleszthetjük úgy, hogy az animáció sebességét is szabályozhassuk egy másik várakozási idő megadásával, bár az AVI fájl ötlete magában foglalja azt, hogy az animáció a saját sebességével fusson.

Kép megjelenítő

Az Image (Kép) komponenst, amelyet már a fejezetben korábban bemutattuk, általában képmegjelenítőnek tekintik. Könnyedén beletölthetünk egy bitmap fájlt (BMP), egy ikont (ICO), vagy egy Windows metafájlt (WMF). A bitmap és ikon fájlok jól ismert fájlformátumok. A Windows metafájlok nem túlságosan

általánosak. A metafájl grafikus parancsokat tartalmaz, hasonlít a GDI függvényhívások listájához, amelyek egy kép újrarajzolásához szükségesek. A metafájlokat általában vektorgrafikának nevezik, és hasonlítanak a clip-art könyvtárak grafikus fájlformátumára.

Megjegyzés

Egy Windows metafájl létrehozásához a programnak meg kell hívnia GDI funkciókat, átirányítva a kimenetüket egy fájlba. Később ez a metafájl lejátszható, vagy elindítható meghívva a megfelelő funkciókat, azaz létrehozva egy grafikát. A metafájlok előnyei: kis tárolóhelyre van szükségük más grafikus formátumokkal szemben, és felépítésük miatt minden eszközön a lehetőségekhez képest szépen jelennek meg

Ha egy teljes kép megjelenítő programot akarunk készíteni (ImageV), az Image komponenst használva, akkor csak egy formra, amely teljesen kitölti a felhasználói felületet, egy egyszerű menüre és egy fájl megnyitó párbeszédablakra lesz szükségünk:

```
object ViewerForm: TViewerForm
  Caption = 'Image Viewer'
  Menu = MainMenu1
  object Image1: TImage
    Align = alClient
  end
  object MainMenu1: TMainMenu
    object File1: TMenuItem...
      object Open1: TMenuItem...
      object N1: TMenuItem...
      object Exit1: TMenuItem...
    object Options1: TMenuItem
      object Stretch1: TMenuItem
      object Center1: TMenuItem
    object Help1: TMenuItem
      object AboutImageViewer1: TMenuItem
  end
  object OpenDialog1: TOpenDialog
    FileEditStyle = fsEdit
    Filter = 'Bitmap (*.bmp)|*.bmp|
      Icon (*.ico)|*.ico|Metafile (*.wmf)|*.wmf'
    Options = [ofHideReadOnly, ofPathMustExist,
      ofFileMustExist]
  end
end
```

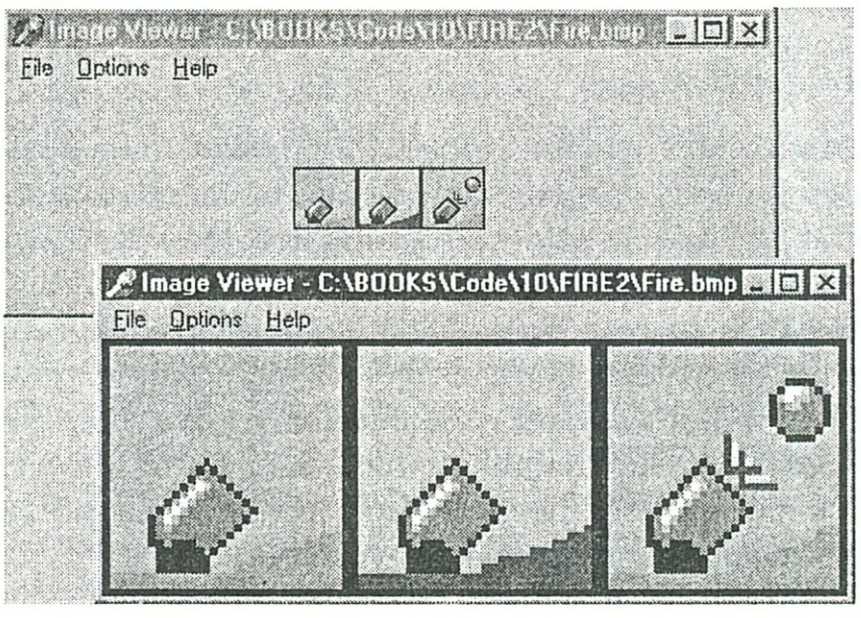
Meglepetésünkre, ez az alkalmazás nagyon kevés programozói munkát igényel, legalábbis az első verzió létrehozásakor. A File menü Exit, és a Help menü About parancsai egyértelműek, a File menü Open parancsának pedig itt a kódja:

```
procedure TViewerForm.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    Image1.Picture.LoadFromFile (OpenDialog1.FileName);
    Caption :='Image Viewer - ' + OpenDialog1.FileName;
  end;
end;
```

A negyedik, és az ötödik menüparancs, az Options menü Stretch és a Center parancsai egyszerűen kapcsolgatják a Stretch (feszítés) vagy Center (középre igazítás) tulajdonságokat (lásd a 11.7. ábrát), és egy pipa jelet adnak magukhoz bekapcsolt állapotban. Itt a Stretch1 menüpont OnClick esemény kezelője:

```
procedure TViewerForm.Stretch1Click(Sender: TObject);
begin
  Image1.Stretch := not Image1.Stretch;
  Stretch1.Checked := Image1.Stretch;
end;
```

Ne felejtsük el, hogy amikor „kifeszítjük" a képet, akkor megváltoztathatjuk a szélesség-magasság arányát, megváltoztatva az alakokat, így nem mindegyik képet lehet megfelelően méretezni. A 256 színű vagy fekete fehér képeken nem mindig működik rendesen a Stretch funkció.



11.7. ábra *Az ImageV program által megjelenített rendes, és széthúzott bitmapek, a Fire2 példából*

Mindezek mellett az alkalmazásnak vannak más hátulütői is. Ha egy megfelelő kiterjesztés nélküli fájlt választunk ki, akkor az Image komponens kivételt vált ki. A rendszer által biztosított kivételkezelő eléggé jól működik: a rossz képfájl nem töltődik be, és a program biztonságosan futhat tovább. Egy másik probléma, ha egy nagy képet töltünk be, a megjelenítőn nincs görgető sáv. Maximális méretre állíthatjuk a megjelenítőt, de ez sem biztos hogy elég lesz. A képmegjelenítő komponensek nem kezelik ezt le automatikusan, de a form igen. A 14. fejezetben a példához hozzáadjuk a gördítősávokat is.

Rajzolás a bittérképbe

Az előző fejezetben említettük, hogy Image komponenst használva közvetlenül rajzolhatunk bitmapbe. Az ablak felülete helyett a memóriába rajzolunk, és onnan másolódik a kép az ablak felszínére. Ennek az előnye az, hogy nem kell minden OnPaint eseménykor újrarajzolni a képet, hanem a komponens másolja vissza a video-memóriába a bitmapet.

Technikailag a TBitmap objektumnak saját rajzfelülete van. Erre rajzolva megváltoztathatjuk a bitmapet. Egy másik megoldás az, ha egy Image komponens felületére rajzolunk, ami egy bitmaphez van kapcsolva, mert így is megváltoztathatjuk a bitmapet. Használhatjuk ezt a megoldást is a tipikus kifestős módszer helyett, ha bármelyik a következő feltételekből teljesül:

- A programnak támogatnia kell a szabadkezes rajzolást, vagy nagyon összetett grafikát (mint például a fraktál képek).
- A programnak nagyon gyorsan kell kirajzolnia több képet.
- A memória (RAM) mérete nagy vagy csak kevés applikációt használunk párhuzamosan.
- Ha lusta programozók vagyunk.

Az utolsó pont érdekes, mert a kifestés általában több kódot igényel, mint a rajzolás, de rugalmasabb is. Egy grafikus programban például, ha kifestést használunk, akkor minden alakzat helyzetét és színét tárolnunk kell. Másvégről pedig könnyedén megváltoztathatjuk egy már meglévő alakzat színét, vagy átmozgathatjuk. Ezek a műveletek nagyon bonyolultak a kifestős megközelítéssel, és a kép mögötti terület elvesztését is okozhatja. Ha egy komplex grafikus alkalmazáson dolgozunk, akkor mindkét megközelítési formát érdemes alkalmaznunk. Ha csak alkalmanként használunk grafikát, akkor a sebességet és a memóriahasználatot kell mérlegeljük, mert a kifestés kevesebb memóriát igényel, viszont a bitmap használatakor programunk gyorsabb.

Alakzatok rajzolása

Most, hogy bemutattuk az Image komponens használatát, ami a bitmapen való festést illeti Delphiben, nézzünk meg egy példát. Mi is lehetne jobb, mint az előző fejezet Shapes példaprogram sorozatának egy újabb verziója? Az ötlet egyszerű. A Shapes példa második verzióját választjuk, amelyben vannak színes alakzatok, de még nem tudjuk elmenteni őket. E program formjára teszünk egy Image komponenst, amely teljesen lefedi a felhasználói felületet, és teljesen átirányítjuk az összes kimeneti műveletet az Image komponens Canvas részére.

A Shapes5 példánkhoz néhány új menüpontot is hozzáadtunk, hogy elmenthessük a képet egy fájlba, és be is tölthessünk egy már létező bitmapet. Ennek érdekében a formhoz hozzáadtunk néhány alapvető párbeszédablakot, mint az OpenDialog, és a SaveDialog. A form háttérszín tulajdonságát meg kellett változtatnunk. Valójában, ha először végzünk egy képen grafikus műveletet, akkor létrejön egy bitmap, amelynek alapértelmezés szerint fehér lesz a háttérszíne. Ha a formnak szürke a háttérszíne, minden alkalommal mikor az ablak újrarajzolódik, zavaró villogás figyelhető meg. Ezért választottunk a formnak is fehér hátteret.

Ennek a példának is meglehetősen egyszerű a kódja, figyelembe véve a műveletek, és a menü parancsok számát. A rajzolási rész lineális és nagyon hasonlít a Shapes2-re, kivéve, hogy az egér események most a képhez kapcsolódnak és nem a formhoz, és a kép felületét használjuk:

```
procedure TShapesForm.Image1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y:
Integer);
begin
  {draw a form in x:y, using the current radius}
  if Button = mbLeft then
    if ssShift in Shift then
      Image1.Canvas.Rectangle (
        X - Radius, Y - Radius,
        X + Radius, Y + Radius)
    ...
```

Az összetett fájlkezelés elkerülése végett a File menü Load, és Save As parancsait valósítottuk csak meg a Save kezelése nélkül, amely általában bonyolultabb. Ehhez egy Change (változás) mezőt kellene adnunk a formhoz, ami jelzi, ha a kép megváltozott, és amelynek értékét többször is ellenőrizzük (mielőtt a megerősítést kérnénk a felhasználótól). Teljes fájlmentési funkció hozzáadásához a 9. fejezetbeni RichNot2-ben található kódot kell beillesztenünk.

A File menü New menü parancsának az Onclick esemény kezelője meghívja a FillArea metódust, ami egy nagy fehér téglalapot rajzol az egész bittérképre. Ebben a kódban láthatjuk a Changed mező használatát is:

```
procedure TShapesForm.New1Click(Sender: TObject);
var
  Area: TRect;
  OldColor: TColor;
begin
  if not Changed or (MessageDlg (
  'Are you sure you want to delete the current image?',
   mtConfirmation, [mbYes, mbNo], 0) = idYes) then
    begin
      {repaint the surface, covering the whole area,
      and resetting the old brush}
      Area := Rect (0, 0, Image1.Picture.Width,
        Image1.Picture.Height);
      OldColor := Image1.Canvas.Brush.Color;
      Image1.Canvas.Brush.Color := clWhite;
      Image1.Canvas.FillRect (Area);
      Image1.Canvas.Brush.Color := OldColor;
      Changed := False;
    end;
end;
```

Természetesen a kódnak el kell mentenie az eredeti színt, és vissza is kell állítani később majd. A színek újrarendezése is szükséges a File menü Load parancsára válaszoló metódusban. Amikor betöltünk egy új bitmapet, akkor valójában az Image komponens létrehoz egy új Canvast az alapértelmezés szerinti tulajdonságokkal. Ezért a program elmenti a toll színét és méretét, és később átmásolja az új Canvasra:

```
procedure TShapesForm.Load1Click(Sender: TObject);
var
  PenCol, BrushCol: TColor;
  PenSize: Integer;
begin
  if not Changed or (MessageDlg (
  'Are you sure you want to delete the current image?',
  mtConfirmation, [mbYes, mbNo], 0) = idYes) then
    if OpenDialog1.Execute then
    begin
      PenCol := Image1.Canvas.Pen.Color;
      BrushCol := Image1.Canvas.Brush.Color;
      PenSize := Image1.Canvas.Pen.Width;
      Image1.Picture.LoadFromFile
            (OpenDialog1.Filename);
      Image1.Canvas.Pen.Color := PenCol;
      Image1.Canvas.Brush.Color := BrushCol;
      Image1.Canvas.Pen.Width := PenSize;
      Changed := False;
    end;
end;
```

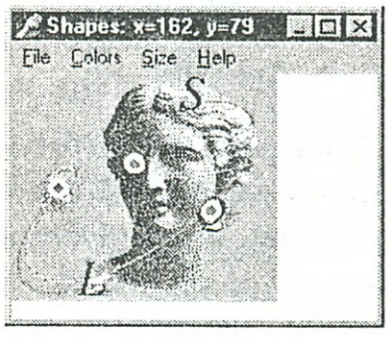
Az aktuális kép elmentése egyszerűbb:

```
procedure TShapesForm.Saveas1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then
  begin
    Image1.Picture.SaveToFile (
      SaveDialog1.Filename);
    Changed := False;
  end;
end;
```

Végezetül itt a form OnCloseQuery eseményének kódja, amely használja a Changed mezőt:

```
procedure TShapesForm.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if not Changed or (MessageDlg (
  'Are you sure you want to delete the current image?',
  mtConfirmation, [mbYes, mbNo], 0) = idYes)
  then
    CanClose := True
  else
    CanClose := False;
end;
```

A Shapes5 egy érdekes program (11.8. ábra), egyszerű, de működő fájltámogatással. Az igazi probléma az, hogy az Image komponens a saját méretével megegyező bitmapet hoz létre. Amikor növeljük az ablak méretét, akkor az Image komponens is átméreteződik, de a bitmap a memóriában nem. Ezért nem rajzolhatunk az ablak jobboldali, és alsó felébe. Van többféle megoldás is: Használjuk a Windows wmGetMinMaxInfo üzenetet, hogy beállítsuk a form maximális méretét (ahogy azt a 10. fejezetben bemutattuk), vagy használjunk állandó határokat, láthatóan megjelölve a rajzolásra használható területet, és így tovább. Úgy döntöttem, hogy így hagyom a programot, ahogy van, mivel a képbe való rajzolás bemutatására így is megfelel.



11.8. ábra *A Shapes5-tel betölthetünk már létező bitmapet, rajzolhatunk rá, és visszamenthetjük a lemezre*

Grafikus listák

Általánossá vált a Windows alkalmazásokban a grafikus listapanelek megjelenése, különösen a fájlok listázásánál. Hogy a Windows 3.1-ben az általános listapanelbe rajzolhassunk, owner-drawnnak kellett deklarálnunk és kifestő kódot kellett biztosítanunk. A Windows 95 bemutatott néhány új általános kontrolt, mint például a ListView és a TreeView, amelyek képesek grafikus listákat és hierarchikus információkat megjeleníteni. Delphiben még mindig megvan az „owner-drawn" listapanel, de általában úgyis a Windows 95 általános kontroljait fogjuk használni. Választási lehetőségként van még a Delphiben néhány eredeti komponens, mint az Outline, a FileListBox, és a DirectoryListBox.

Ha az általános komponenseket alkalmazzuk az alkalmazásunkban, akkor a felhasználók már tudni fogják, hogy hogyan kell használni őket, és nagyon modern felhasználói felületűnek fogják tartani a programunkat. A TreeView és a ListView a két kulcsfontosságú komponense a Windows 95 Intézőnek, és feltételezhetjük, hogy sok felhasználó jártas lesz a használatukban. Az általános kontrolok hátulütője az, hogy nem tudjuk őket lefordítani a Delphi 16-bites verziójával, és futtatni sem tudjuk a Windows 16-bites verzióin. Az Outline komponenst használva ezek helyett, az alkalmazásunk átvihető lesz a Delphi különböző verziói között, és a Windows 16-bites és 32-bites verziói között.

Ahogy már említettük, mutatunk egy egyszerű owner-drawn listapanelt, összehasonlító példaként a TreeView és az Outline használatával, és egyet, amely a ListView komponenssel foglalkozik. Az itt látható példák meglehetősen egyszerűek: Ezeket a komponenseket fogjuk használni összetettebb példák létrehozására is a későbbi fejezetekben.

Színek listájának kirajzolása

Ahogy azt már a menüknél láthattuk, a listapaneleknek van egy owner-drawn képességük, amely azt jelenti, hogy egy programmal festhetjük meg a listapanel elemeit. Amíg a Delphi korlátozott támogatást biztosít az owner-drawn menükhöz, Windows API hívásokra, és a Windows üzenetek közvetlen kezelésére kötelezve minket, a TListbox osztály (mint a TCombobox osztály) közvetlen támogatást nyújt mindehhez, egyszerűbbé téve a kódot.

Owner-drawn listapanel létrehozásához először be kell állítanunk a Style tulajdonságát lbOwnerDrawFixed-re, vagy lbOwnerDrawVariable-ra. Az első érték jelzi, hogy majd megadjuk a listapanel elemeinek magasságát az ItemHeight tulajdonság beállításával, és ez lesz a magassága minden egyes elemének. A második owner-draw stílus jelzi, hogy a listapanel különböző magasságú elemeket tartalmaz. Ebben az esetben a komponens meghívja az OnMeasureItem eseményt minden egyes elemhez, hogy megtudja az elemek magasságait.

Az ODList példában, az első, egyszerűbb megközelítéssel kezdjük. A példa célja szín információk tárolása a listapanel elemeivel, és az elemeket színesen kirajzolni (több színnel, egy szín helyett). Itt vannak a példa fő formján lévő komponensek tulajdonságai:

```
object ODListForm: TODListForm
  Caption = 'Owner-draw Listbox'
  Font.Color = clBlack
  Font.Height = -32
  Font.Name = 'Arial'
  Font.Style = [fsBold]
  Position = poDefault
  OnCreate = FormCreate
  TextHeight = 37
  object ListBox1: TListBox
    Align = alClient
    Style = lbOwnerDrawFixed
    OnDblClick = ListBox1DblClick
    OnDrawItem = ListBox1DrawItem
  end
  object ColorDialog1: TColorDialog...
end
```

Ahogy láthatjuk, beállítottunk egy nagy betűtípust a formnak, és az lbOwnerDrawFixed értéket a listapanel Style tulajdonságának. A form TextHeight tulajdonsága megfelel a megjelenített szöveg magasságának pixelben mérve. Ezt az értéket kell használnunk a listapanel ItemHeight tulajdonságánál. Jobb megoldás, ha futási idő alatt számítjuk ki ezt az értéket, mert ha később szerkesztési idő alatt megváltoztatjuk a fontot, akkor nem kell beállítgatnunk az elemek magasságát annak megfelelően.

Megjegyzés

Éppen most írtuk le a TextHeight-et mint a form egy attribútumát, nem pedig egy tulajdonságát. És valójában ez nem egy tulajdonság, megkérdezhetnék, hogy akkor a Delphi hogy menti ezt el a DFM fájlba? A válasz a Delphi ún. folyam (stream) mechanizmusa a tulajdonságokon alapul, valamint különleges tulajdonság-klónokon, amelyeket a DefineProperties metódus hoz létre. Ez tényleg egy haladó téma, és nem térünk ki rá a könyvben. Információkat találhatunk erről a Delphi Súgó fájljaiban, vagy haladó Delphi könyvekben (a Delphi Developer's Handbook-ot is beleértve).

Mivel a TextHeight nem egy tulajdonság, ámbár a form leírásánál szerepel, nem érhetjük el közvetlenül. Tanulmányozva a VCL forráskódját, megtaláltuk, hogy ez az érték a form GetTextHeight metódusának meghívásával számítódik ki. Mivel ez privát érték, nem hívhatjuk meg ezt a funkciót. Amit viszont megtehetünk az, hogy bemásoljuk a kódot (amely meglehetősen egyszerű) a form FormCreate metódusába:

```
ListBox1.ItemHeight := Canvas.TextHeight('0');
```

A következő lépésben a listapanelhez elemeket kell hozzáadnunk. Mivel ez a színek listapanelje, ezért a színek neveit adjuk hozzá a listapanel elemeihez, és megfelelő szín értékeket az elemekhez tartozó Objects adattárhoz Ahelyett, hogy a két értéket külön adjuk hozzá, írtunk egy eljárást, ami hozzáadja az új elemeket a listához:

```
procedure TODListForm.AddColors (Colors: array of
    TColor);
var
  I: Integer;
begin
  for I := Low (Colors) to High (Colors) do
    ListBox1.Items.AddObject (
      ColorToString (Colors[I]),
      TObject(Colors[I]));
end;
```

Ez a metódus egy nyílt-tömb paramétert használ, amely technikát a 4. fejezetben tárgyaltuk. Minden egyes elemhez, paraméterként átadva a szín nevét hozzáadjuk a listához, és az értékét a hozzátartozó adatnak az AddObject metódust meghívva. A színnek megfelelő stringet megkapjuk a Delphi ColorToString funkcióját meghívva, amely visszaadja a stringben a megfelelő szimbolikus szín konstanst, ha van , vagy egy stringet amiben a szín hexadecimális értéke van. A színadat hozzáadódik a listapanelhez miután az értékét átadjuk a TObject adat típusnak (4 byte hosszú hivatkozás), amely szükséges az AddObject metódusnak.

Tipp

A ColorToString mellett, amely egy szín értékét konvertálja a megfelelő szöveggé, vagy hexadecimális értékké, van egy Delphi funkció, ami egy megfelelően formázott szöveget konvertál színné, ez a StringToColor.

Az ODList példában ezt a metódust a form OnCreate eseménykezelőjében hívjuk meg (miután beállítottuk az elemek magasságát):

```
procedure TODListForm.FormCreate(Sender: TObject);
begin
  ListBox1.ItemHeight := Canvas.TextHeight('0');
  AddColors ([clRed, clBlue, clYellow,
              clGreen, clFuchsia, clLime,
    clGray, RGB (213, 23, 123), RGB (0, 0, 0)]);
end;
```

A kód amely kirajzolja az elemeket nem igazán összetett. Egyszerűen visszanyerjük az elemmel társított színt, beállítjuk mint a font színét, és kiírjuk a szöveget:

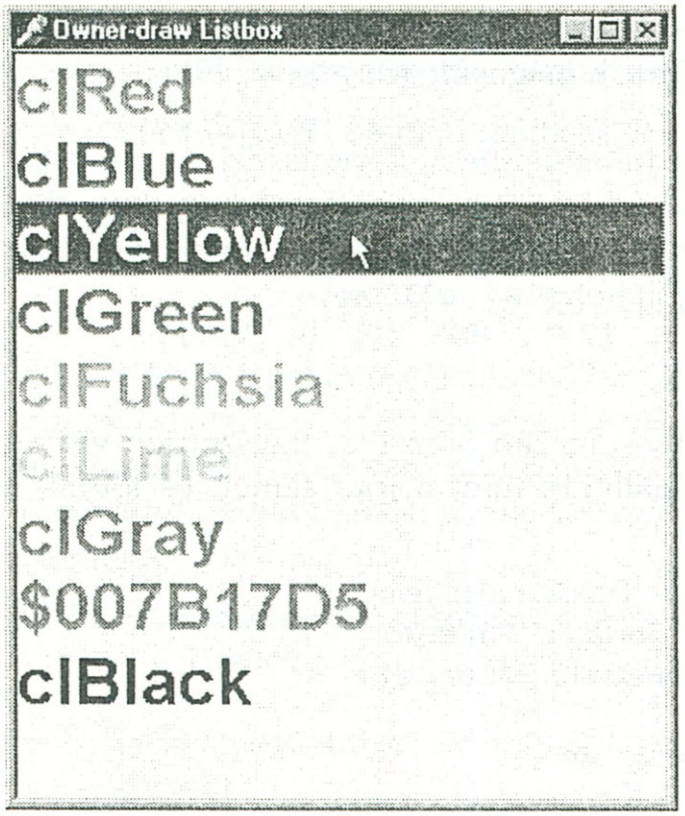
```
procedure TODListForm.ListBox1DrawItem(
  Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
begin
  with Control as TListbox do
  begin
    // erase
    Canvas.FillRect(Rect);
    // draw item
    Canvas.Font.Color:=TColor (Items.Objects [Index]);
    Canvas.TextOut(Rect.Left, Rect.Top,
       Listbox1.Items[Index]);
   end;
end;
```

A 11.9. ábrán láthatunk egy példát a program indítás utáni kimenetéről. A példában új elemeket is hozzáadhatunk, a listapanelre duplán kattintva:

```
procedure TODListForm.ListBox1DblClick(Sender: TObject);
begin
  if ColorDialog1.Execute then
    AddColors ([ColorDialog1.Color]);
end;
```

Ha kipróbáljuk ezt a képességet, akkor észrevehetjük, hogy néhány színnél, amelyet hozzáadtunk a szín neve jelenik meg (a Delphi szín konstans neve), míg másoknál egy hexadecimális szám. Ugyanez igaz a FormCreate metódusban az utolsó két hozzáadott elemre is. Ahogy azt a 11.9. ábrán is láthatjuk, az egyikük egy szám, a másikuk egy konstans név.

Ámbár egy owner-drawn listapanelt használni meglehetősen egyszerű, Delphi biztosít olyan komponenseket, amelyek implementálják ezt a képességet, és más fejlettebbeket is. Főként az Outline és a TreeView komponensek használnak grafikát többszintű listák felépítésére.



11.9. ábra *Az ODList példa színezett, owner-draw listája*

A könyv rövid áttekintése

Először bemutatjuk az Outline (vázlat) komponens néhány sajátságát. Az Outline komponenssel a listaelemek szövege elé bitmapeket tehetünk, és az elemek grafikus hierarchiáját építhetjük fel vele. Egy egyszerű elemekből álló lista létrehozásához csak hozzá kell adnunk őket a Lines tulajdonsághoz, és adnunk kell egy saját bitmapet a PictureLeaf tulajdonsághoz.

Az Outline használata áttekinthető képet ad, ha hierarchikus információkat kell megjelenítenünk. Példaként ennek a könyvnek az előző kiadásából a tartalomjegyzéket használjuk a BookOut példa megírására.

Figyelem!

Amikor elkezdtem megírni a példát, akkor Outline-nak akartam nevezni. Miután megpróbáltam, észrevettem, hogy a Pascal nem tudott megfelelően hivatkozni a rendszer Outline egységére, mert azt egy program fájl neve rejtette. A történet mondanivalója az, hogy kerüljük az ismételt azonosítók használatát általában, különösen az egység és modul nevek használatát (a VCL-ben lévőkét is beleértve).

Mielőtt a példát néznénk, elmagyarázzuk az Outline komponens néhány tulajdonságát és stílusát. A Lines tulajdonságban van a kontrol tartalmának szöveges változata, és szerkesztési idő alatt elérhetjük. Az Items tulajdonságban ugyanaz az információ van, de hierarchikus struktúrába rendezve, és csak futási idő alatt érhető el.

Amikor beírjuk, vagy beolvassuk egy fájlból egy vázlat szövegét, akkor vagy szóközt, vagy tabulátor-karaktert hagyhatunk egy elem előtt, hogy jelezzük a szint azonosítóját. Szerkesztési idő alatt a tulajdonsághoz kapcsolt String Editorral dolgozhatunk a stringek listáján, és szövegfájlt is betölthetünk.

Tipp

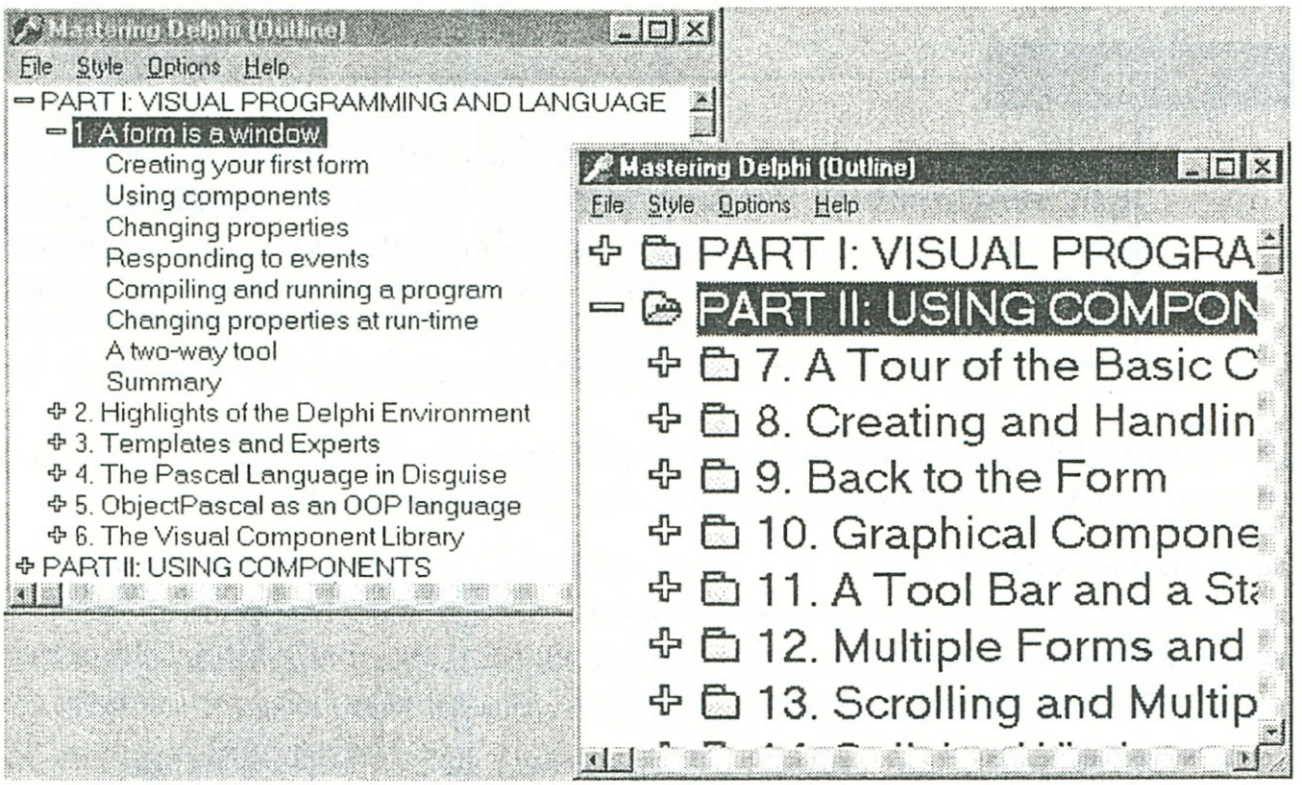
Amikor a Delphi String List Editorát használjuk, akkor a TAB billentyűt megnyomva a fókusz a lejjebb lévő gombok egyikére mozog ahelyett, hogy a szövegbe tenne egy tabulátort. Ha tabulátor karaktert akarunk egy elem elé tenni, akkor használjuk a CTRL+TAB billentyűket.

Ha egyszer definiáltuk egy Outline komponens szövegét, akkor beállíthatjuk a stílusát. Az OutlineStyle tulajdonságnak hat lehetséges értéke van:

- osPictureText definiál egy olyan outline-t, amelyben nyitott mappa kép jelöli a kiterjesztett csomópontokat, és egy zárt mappa kép jelöli a összehúzott csomópontoknál, vagy egy lap ikonnal jelzi az elemeket, amelyeknek már nincsenek alelemei.
- osTreePictureText definiál egy olyan outline-t, amelyben nyitott és zárt képek vannak, és vonalak egészítik ki, hogy kiemeljék az elemek elhelyezkedését a különböző szinteken.
- osPlusMinusText definiál egy olyan outline-t, amelyben csak plusz és mínusz képek vannak a mappák helyett.
- osPlusMinusPictureText definiál egy olyan outline-t, amelyben nyitott és zárt képek mellett plusz és mínusz képek is vannak.
- osText definiál egy olyan outline-t, amelyben csak szöveg látható ikonok nélkül.
- osTreeText definiál egy olyan outline-t, amelyben csak a grafikus fa van a szövegekkel, bitmapek nélkül.

A BookOut példa futtatásakor láthatjuk ezen tulajdonságok működését. Az outline mindegyik verziója mutatja a szöveget a megfelelő azonosítással. A másik különös tulajdonság az Options, amellyel megadhatjuk a fa gyökerének

megjelenítését (ooDrawTreeRoot), választhatjuk a fókuszt jelző keret bekapcsolását (ooDrawFocusRect), vagy méretre igazíthatjuk az általános bitmapeket, hogy a mérete illeszkedjen az elem méretéhez (ooStretchBitmap), amely a komponens betűtípus méretétől függ. Ha nem állítjuk be az ooStretchBitmap opciót, akkor bitmapek eredeti méretükben jelennek meg kivágva egy nagy részt, vagy nagy helyet hagyva maguk körül. Amikor nagy fontot, vagy nagy bitmapet választunk ez az opció még hasznos lehet. Ezeket a tulajdonságokat megváltoztathatjuk futási idő alatt a BookOut példát használva, ahogy azt a 11.10. ábrán is láthatjuk.



11.10. ábra *A BookOut programban az Outline sok tulajdonságát beállíthatjuk, ahogy azt az itt látható két példát összehasonlítva láthatjuk is*

Az Outline komponens fő elemeit átvéve már rátérhetünk a példára, amely bemutatja a működését. A BookOut példa lehetővé teszi az outline különböző stílusainak tesztelését, amelyeket az alkalmazás menüjében tudunk állítani. A BookOut példa egyszerű formja egy outlinet tartalmaz, ami kitölti a felhasználói felület egészét, és egy menüt:

```
object Form1: TForm1
  Caption = 'Mastering Delphi (Outline)'
  Menu = MainMenu1
  object Outline1: TOutline
    Lines.Nodes = (
```

```
        'PART I: VISUAL PROGRAMMING AND LANGUAGE'
        #9'1. A form is a window '
        #9#9'Creating your first form'
        #9#9'Using components'
        // and so on... )
      Options = []
      Align = alClient
      BorderStyle = bsNone
      ItemSeparator = '\'
    end
    object MainMenu1: TMainMenu
      object File1: TMenuItem...
        object Exit1: TMenuItem...
      object Style1: TMenuItem...
        object PictureText1: TMenuItem
          Caption = '&Picture Text'
          GroupIndex = 1
          RadioItem = True
          OnClick = PictureText1Click
        end
        object PlusMinusPictureText1: TMenuItem...
        object PlusMinusText1: TMenuItem...
        object Text1: TMenuItem...
        object TreePictureText1: TMenuItem
          Caption = 'T&ree Picture Text'
          Checked = True
          GroupIndex = 1
          RadioItem = True
          OnClick = TreePictureText1Click
        end
        object TreeText1: TMenuItem...
      object Options1: TMenuItem...
        object DrawTreeRoot1: TMenuItem...
        object DrawFocusRect1: TMenuItem...
        object StretchBitmaps1: TMenuItem...
        object N1: TMenuItem...
        object Font1: TMenuItem
      object Help1: TMenuItem...
        object About1: TMenuItem...
    end
    object FontDialog1: TFontDialog...
  end
```

Az alkalmazás kódja is egyszerű. Csak az outline stílusait be-, és kikapcsoló parancsokat tartalmaz, amelyek jelölőnégyzetek, vagy rádiógombok formájában jelennek meg a kiválasztott menüponton belül. A Style menü elemei kölcsönösen egyediek és rádió menüelemekből állnak. Az Onclick eseménykezelők kódja nagyon egyszerű, ahogy azt a következő metódusban is láthatjuk:

```
procedure TForm1.PictureText1Click(Sender: TObject);
begin
  Outline1.OutlineStyle := osPictureText;
  PictureText1.Checked := True;
end;
```

Az Options legördülő menü elemei ehelyett egymástól függetlenül működnek. Nem különböző értékeknek felelnek meg, hanem egy halmaz elemeit alkotják, ezért ellenőrizni azt, hogy be vannak-e kapcsolva, és be-, vagy kikapcsolni őket több műveletet igényel:

```
procedure TForm1.DrawTreeRoot1Click(Sender: TObject);
begin
  if ooDrawTreeRoot in Outline1.Options then
   Outline1.Options:=Outline1.Options-[ooDrawTreeRoot]
  else
   Outline1.Options:=Outline1.Options+[ooDrawTreeRoot];

  DrawTreeRoot1.Checked := not DrawTreeRoot1.Checked;
end;
```

A legördülő menü utolsó menüpontjával állíthatunk be új fontot, amit kiválasztva a program elindít egy FontDialog komponenst, szokás szerint.

Egy megjegyzés: az outline bitmapjait kis formon kifeszítve furcsa eredményt kapunk a Delphinek ebben a verziójában.

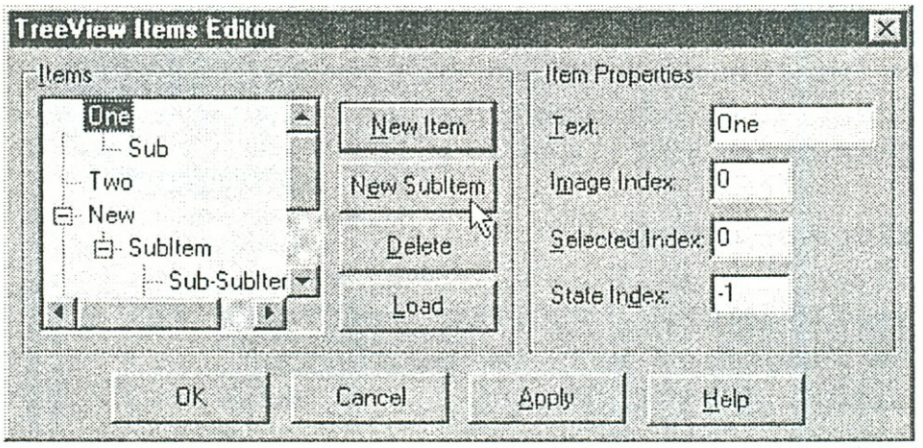
A fejezetek struktúrája

Most már megpróbálhatunk készíteni egy hasonló példát a TreeView komponenst használva az Outline komponens helyett. Nincs közvetlen módja, hogy szövegfájlt töltsünk be ebbe a komponensbe, tehát ezt magunknak kell megírnunk. A TreeView-nak rugalmas a felhasználói felülete (szerkesztés és elemek megragadásának támogatásával), de egyben általános is, hiszen például ez a Windows Intéző felhasználói felülete is. Számos tulajdonsága van, és sokféleképpen lehet beállítani a sorok, vagy sor típusok bitmapeit. A TreeView valójában egy TImageList komponenshez van kapcsolva. Ez lehetővé teszi a programozó számára, hogy több bitmapet társítson a TreeView-val, és hogy egy indexet rendeljen minden csomóponthoz, vagy elemhez.

Az első példa, amely TreeView-t használ, nagyon hasonló lesz az előzőhöz. Az új példa neve ChapTree, ami egy formon alapul, amelyen van egy TreeView, ami az egész felhasználói területhez van igazítva, és egy menü néhány megfelelő paranccsal, amelyekkel a megfelelő tulajdonságokat ki-, és bekapcsolhatjuk. Van egy parancs a font megváltoztatására is:

```
object Form1: TForm1
  Caption = 'Chapters Tree'
  Menu = MainMenu1
  OnCreate = FormCreate
  object TreeView1: TTreeView
    Align = alClient
  end
  object MainMenu1: TMainMenu
    object File1: TMenuItem...
      object Exit1: TMenuItem...
    object Options1: TMenuItem...
      object Root1: TMenuItem...
      object Buttons1: TMenuItem...
      object Lines1: TMenuItem...
      object N1: TMenuItem...
      object Font1: TMenuItem...
    object Help1: TMenuItem...
      object About1: TMenuItem...
  end
  object FontDialog1: TFontDialog...
end
```

Ebben a példában betölthettük volna a TreeView csomópontjainak struktúráját szerkesztési idő alatt, ahogy azt az előző példában is tettük, hála a Delphi 3 TreeView Items tulajdonságához hozzáadott szerkesztőnek (11.11. ábra), amelyben felépíthetjük az adatok struktúráját. Most azonban úgy döntöttünk, hogy a program indulásakor töltjük be a TreeView adatokat.



11.11. ábra *A TreeView Items tulajdonságának szerkesztője*

Az új komponens Items tulajdonságának sok beépített funkciója van, amellyel átalakíthatjuk a stringek hierarchiáját. Például építhetünk egy kétszintű fát a következő sorokkal:

```
var
  Node: TTreeNode;
begin
  Node := TreeView1.Items.Add (nil, 'First level');
  TreeView1.Items.AddChild (Node, 'Second level');
```

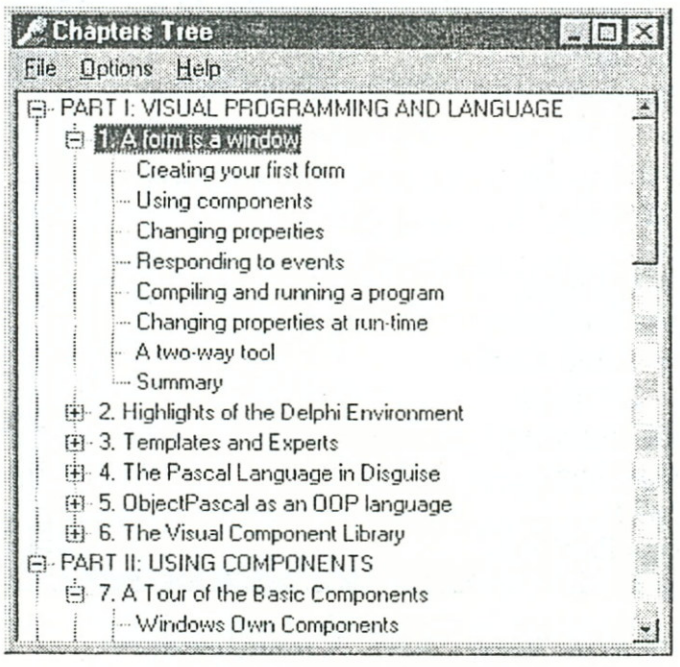
Az Add, és az AddChild metódusokat használva futási idő alatt felépíthetünk egy komplex struktúrát. De hogyan töltjük be az információt? Egy fájl közvetlen elérése helyett (ami nem nehéz egyébként) létrehozunk egy Memo komponenst futási idő alatt, betöltünk egy text fájlt, az outline elmentett formáját az előző példából, és elemezzük. Alapvetően a program megszámolja a tabulátor-karaktereket (#9), és hozzáad egy új elemet a fához a megfelelő szinten. Ha nincs tabulátor, akkor az új elemet a gyökérhez kapcsolja, máskülönben az előző szinten lévő utolsó elemhez kapcsolja. Bonyolultnak hangzik? A kód valójában nem túl tekervényes:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Node, Node1, Node2: TTreeNode;
  Memo: TMemo;
  I: Integer;
begin
  // initialize node variables (just in case)
  Node := nil;
  Node1 := nil;
  Node2 := nil;
  // create a temporary memo control
  Memo := TMemo.Create (self);
  try // finally: memo should be destroyed
    Memo.Parent := self;
    Memo.Width := 1000;
    try // except file not found
      Memo.Lines.LoadFromFile ('toc.txt');
    except
      on E: EFOpenError do
      begin
        MessageDlg ('File "TOC.TXT" not found'#13 +
          'The ChaptTree will be terminated',
          mtError, [mbOK], 0);
        // terminate the program
        Halt;
      end; // on E: EFOpenError do
    end; // except

    for I := 0 to Memo.Lines.Count - 1 do
      if Memo.Lines[I][1] <> #9 then
        Node := TreeView1.Items.Add (nil,
          Trim(Memo.Lines[I]))
```

```
        else if Memo.Lines[I][2] <> #9 then
          Node1 := TreeView1.Items.AddChild (Node,
            Trim (Memo.Lines[I]))
        else if Memo.Lines[I][3] <> #9 then
          Node2 := TreeView1.Items.AddChild (Node1,
            Trim (Memo.Lines[I]))
        else if Memo.Lines[I][4] <> #9 then
          TreeView1.Items.AddChild (Node2, Trim
            (Memo.Lines[I]));
    finally
      Memo.Free;
    end;
  end;
```

Ahogy láthatjuk, a `try-except` részt akkor használtuk, ha nem találtuk a fájlt (file not found), a `try-finally` részt pedig az ideiglenes Memo komponens megsemmisítésének biztosítására alkalmaztuk. A csomópontokhoz rendelt nil értékek főleg a fordítási figyelmeztető üzenetek elkerülésére szolgáltak, de egyben egyfajta „biztonsági háló" is, ha esetleg rossz a kód. A másik figyelemre méltó utasítás a Trim funkció, amellyel levághatjuk az elválasztójeleket, és a tabulátorokat a stringekről. A 11.12. ábrán láthatjuk a program futásának eredményét. Figyeljük meg, hogy a komponens lehetővé teszi a fa csomópontjainak, és a levelek szövegének közvetlen szerkesztését.

Hiba esetén, amikor a program nem inicializálódhat megfelelően (például ha nem találja a TOC.TXT fájlt), akkor azonnal le kell állítanunk a programot, hogy a felhasználónak ne jelenítsünk meg egy üres, és használhatatlan formot. A program megszakítására a Halt globális eljárást hívtuk meg.

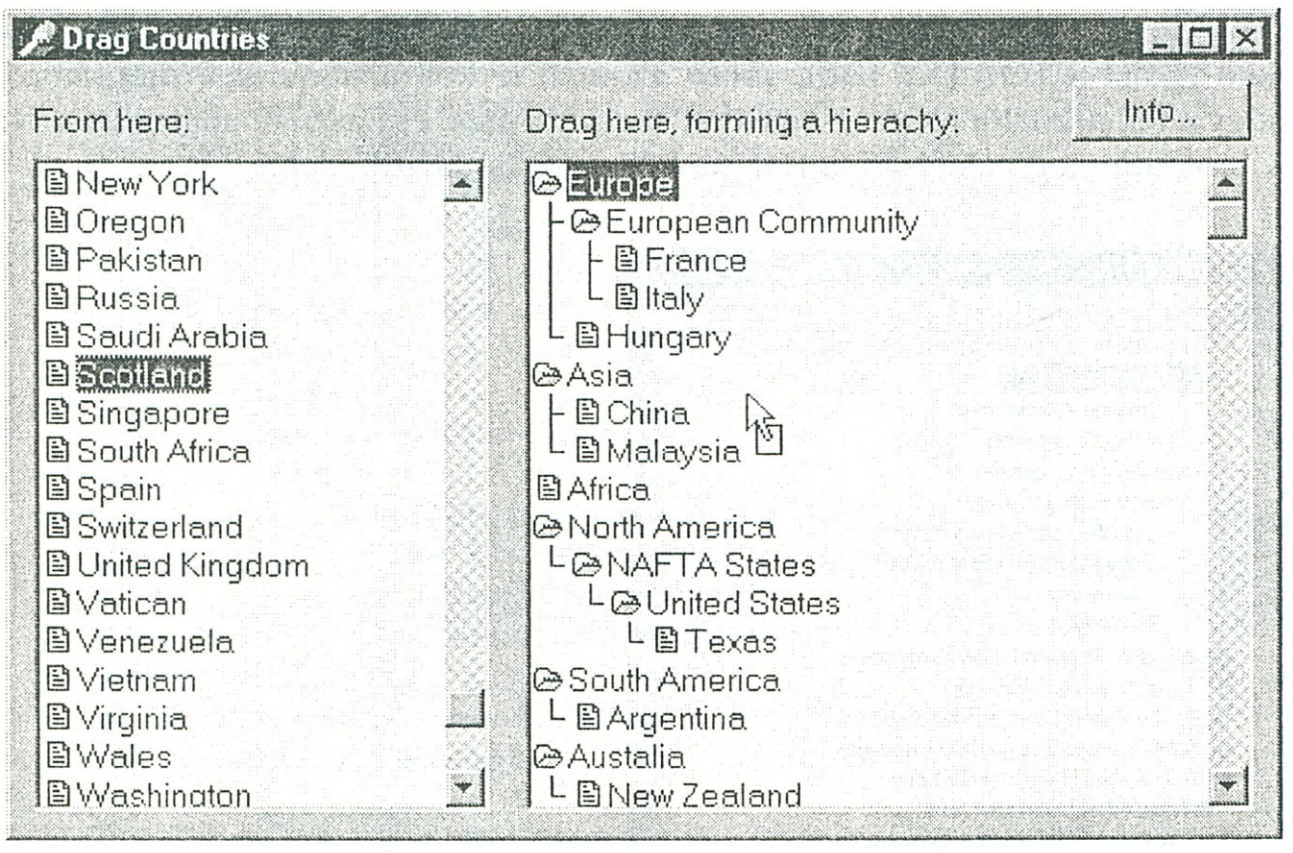


11.12. ábra *A ChapTree példa kimenete*

Az Outline csomópontjai

Ahogy azt a TreeView-nál is láthattuk, az Outline komponens belsejében is van egy objektum lista (TOutlineNode típusú). Az összes objektumnak van szöveges része, valószínűleg valamilyen adat, és egy csomó más tulajdonsága. Például a Level jelzi az elem szintjének azonosítóját, és a FullPath tartalmazza a nevet, a szülő elemeket is beleértve. Ha az Outline komponenst jobban ki akarjuk használni, akkor tanulmányozzuk a csomópont osztályokat (Node classes) részletesebben. Az outlineon végezhető műveletek unalmas ismertetése helyett nézzünk egy kissé fejlettebb példát a komponens használatára.

A példa neve Nodes, és egyfajta földrajzi játék lesz. A program fő formján van két grafikus listapanel (lásd 11.13. ábra). A bal oldali lista néhány országot, államot, és országokat összefogó szervezetet tartalmaz (mint például az Európai Unió). Természetesen a lista nem teljes: ha olyan országban, vagy államban él, amely nincs a listában akkor azt adja hozzá a listához. A jobb oldali lista a hét kontinenst tartalmazza. A játék célja, hogy ragadjunk meg elemeket a bal oldali outlineból, és tegyük át a jobb oldalra a megfelelő helyre a hierarchiában. Hogy ezt elérjük válasszunk ki egy elemet a bal oldalon és húzzuk át jobb oldalon lévő szülő csomópontra. Például válasszuk ki „Brazíliát", és húzzuk át „Dél-Amerikába".



11.13. ábra *A Nodes program az Outline komponens használatát bemutató fejlesztett példa*

Más elemeket előfordulhat, hogy alacsonyabb szintre kell áthúznunk. Például Angliát az Egyesült Királyság alá kell húzni, ami pedig az Európai Unió alá kerül, ez pedig Európa alá kerül. Amikor befejeztük, akkor információt kaphatunk az elemről az Info gombot megnyomva. Ez az információ tartalmazza az elem teljes útvonalát, ahogy azt a 11.14. ábrán is láthatjuk.

11.14. ábra *A „United Kingdom" elem információi, a teljes elérési útvonalával együtt, ahogy azt a Nodes példa kijelzi*

A példa formjának elkészítése nagyon egyszerű: csak két Outline komponens, két címke, és egy gomb kell hozzá. A két Outline-nak nincsenek különleges tulajdonságai. A gyökér csomópontot eltávolítottuk, és szöveget adtunk hozzá az elemekhez. Figyeljük meg, hogy az automatikus áthúzás le van tiltva, helyette a manuális áthúzást használtuk.

Tipp

Ha automatikus húzást választunk egy listapanelnél, vagy valami hasonló komponensnél, akkor a programunknak kell kezelnie az elemek kiválasztását. Ahhoz, hogy egy felhasználó kiválaszthasson elemeket, és húzhassa is őket, manuális húzást kell használnunk, a BeginDrag metódust meghívva, és a hamis (false) paramétert kell átadnunk. Ebben az esetben a vonszolás nem kezdődik meg azonnal, és a kattintás tulajdonság kiválaszt egy új elemet.

A manuális áthúzást egyszerű kezelni. Amikor a felhasználó megnyomja a bal egérgombot az első outline komponens felett, akkor (hacsak az nem üres) a program meghívja a BeginDrag metódust:

```
procedure TForm1.Outline1MouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
```

```
    if (Outline1.ItemCount > 0) and (Button = mbLeft)
  then
      OutLine1.BeginDrag (False);
  end;
```

Amint a gombot elengedtük, a program automatikusan meghívja az első outline EndDrag metódusát. A második Outline komponens definiál egy egyszerű eseménykezelőt, az OnDragOver-t, és végrehajtja az igazi munkát az OnDragDrop eseményben:

```
procedure TForm1.Outline2DragDrop(
  Sender, Source: TObject; X, Y: Integer);
var
  Current: Integer;
begin
  Current := Outline2.GetItem (X, Y);
  if Current > 0 then
  begin
    Outline2.AddChild (Current, Outline1.
      Lines[Outline1.SelectedItem - 1]);
    Outline2.Items [Current].Expanded := True;
    Outline1.Delete (Outline1.SelectedItem);
  end
  else
    MessageDlg ('You''ve not dragged over an item',
      mtError, [mbOk], 0);
end;
```

Ez az utolsó metódus, amely a program lelke, meglehetősen komplex. Amikor a felhasználó áthúz egy új elemet, akkor a program először meghatározza a cél outline elemét, amelyre az áthúzott elemet ledobtuk, a GetItem funkciót és az esemény által átadott koordinátákat használva. Ezután a program kiválasztja ezt az elemet, mint az outline aktuális elemét, amelyre vonatkozni fog az AddChild metódus meghívása. A -1 szükséges, mert a Lines tömb indexe nullától indul, amíg az elemek 1-től vannak sorszámozva. Ki lehet nyerni az elem szövegét közvetlenül a Lines tömbből, mert a forrás lista elemeinek nincs azonosítójuk. A Lines értéke egy azonosítós elemnél valójában tartalmazhat tabulátor karaktert az elején. A Windows ezt egy függőleges vonalként jelzi ki (ebben az esetben a Trim funkcióval kicselezhetjük). Miután az elemet bemásoltuk a cél outlineba, ki kell törölnünk a forrás outlineból, és a szülő csomópontját ki kell terjeszteni, hogy látsszon az új elem.

Az egyetlen másik bonyolult metódus az, amelyik kiszámolja azt a stringet, amelyet információként jelenítünk meg (11.13. ábra). Ez a string magába foglalja az elem teljes útvonalát, szintjét, és azt, hogy köztes csomópont-e:

```
procedure TForm1.ButtonInfoClick(Sender: TObject);
var
  Node: TOutlineNode;
  Text: string;
begin
  Node := Outline2.Items [Outline2.SelectedItem];
  Text := Format ('Item: %s'#13'Level: %d',
    [Node.FullPath, Node.Level]);
  if Node.HasItems then
  begin
    Text := Text + #13'* Has sub-items';
    if Node.Expanded then
      Text := Text + ' and is expanded';
  end;
  MessageDlg (Text, mtInformation, [mbOk], 0);
end;
```

Egy grafikus lista

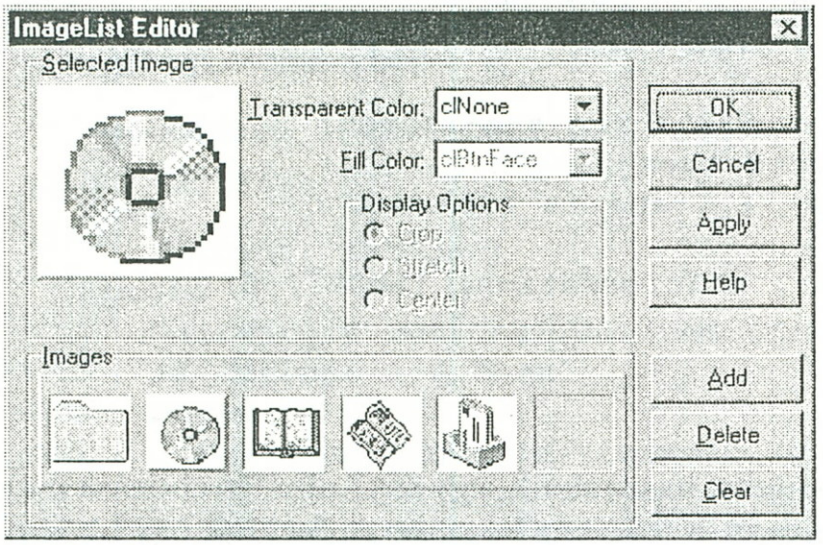
Amikor egy TreeView komponenst használunk, egy struktúra létrehozása mellett minden egyes csomóponthoz és levélhez hozzárendelhetünk egy bitmapet. Valójában megadhatunk bitmapeket, amely jelzi az elem állapotát (például kiválasztott elem), és leírja az elem tartalmát grafikusan. Vegyük figyelembe újra a Windows Intéző alkalmazást: a könyvtárak fastruktúrájában a bal oldali panelen vannak kis képek, amelyek jelzik, hogy a megnyitott és bezárt könyvtárakat (az elem állapota), de vannak speciális bitmapek a Lomtár, a Rendszer, és a Hálózat, és még sok más jelölésére.

Grafikus TreeView példa mutatása helyett egy másik komponenst fogunk használni, a ListView-et, amely egy másik jól használható Windows 95-ös általános kontrol. A két komponensben az a közös, ahogy bitmapeket kapcsolnak az elemekhez, habár a ListView komponens sokkal bonyolultabb, mert kis és nagy bitmapet is kapcsol minden egyes eleméhez.

Hogyan kapcsoljuk a képeket egy listához vagy egy fához? Egy különleges komponenst kell használnunk, a TImageList-et. Egy képlista egyfajta interfész, ahol egy bitmapnek többféle alképe lehet, és mindegyik egy elemhez kapcsolódik a listában. Ha van néhány kész bitmapünk, akkor egy formra tehetünk egy ImageList (képlista) komponenst és elindíthatjuk a kép lista szerkesztőt (Image List Editor) a komponens lokális gyorsmenüjét használva. Ahogy azt a 11.15. ábrán is láthatjuk, ez a speciális szerkesztő lehetővé teszi több kép hozzáadását az ImageList-hez, és néhány tulajdonságuk definiálását.

Mielőtt használnánk a képlista szerkesztőt, be kell állítanunk a lista bitmapjeinek a magasságát, és a szélességét a Width, és a Height tulajdonságokkal (ez egy nem látható komponens, tehát nem használ magától mérettel kapcsolatos tulajdonságokat). Fontos, hogy ezt még az elején megtegyük, mert ha később

tennénk, akkor elveszítjük a lista elemeit figyelmeztetés nélkül. Ha beállítottuk a méretet, akkor a lista összes képe ekkora lesz. A RefList példa ImageList1 komponensének felépítésekor használtuk ezt a módszert, miután beállítottuk a szélességet és a magasságot 32-re. Ezt a képlistát, valójában a ListView komponens LargeImages tulajdonságához használjuk.



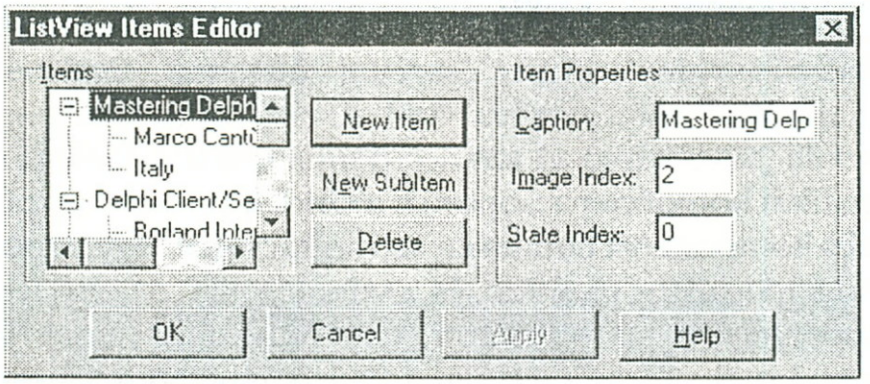
11.15. ábra *Az ImageList Editorban létrehozhatunk egy listát, és hozzáadhatunk azonos méretű bitmapeket*

A SmallImages tulajdonság megadásához egy másfajta megközelítést használtunk: létrehoztunk egy önálló bitmapet (16×80 pixel), amelyben az összes kép benne van. Ezután hozzáadtuk a bitmapet egy erőforrás fájlhoz, és írtunk egy rövid kódot, amely betölti az összes képet egyszerre (nem egyesével). Létrehoztunk egy második ImageList komponenst futási idő alatt (a szülőnek a formot adva meg – lásd a Create konstruktor paraméterét – így nem kell manuálisan megsemmisíteni a befejezéskor). Íme a form OnCreate eseménykezelőjének a kódja:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  ImageList2: TImageList;
begin
  ImageList2 := TImageList.Create (self);
  ImageList2.ResourceLoad(rtBitmap,
    'SmallImages', clOlive);
  ListView1.SmallImages := ImageList2;
  ListView1.Arrange (arDefault);
end;
```

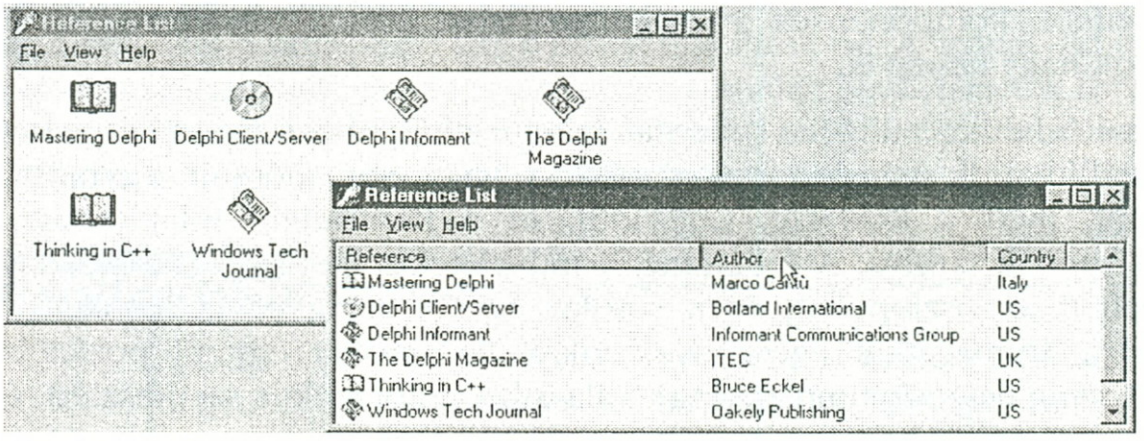
A két ImageList definíciója után hozzáadtunk a listához néhány elemet az Items tulajdonság szerkesztőjével. Definiálhatunk elemeket, és úgynevezett

„al-elemeket", amelyek csak a részletes nézetben láthatóak (amikor beállítjuk a ViewStyle tulajdonságnál a vsReport értéket). Minden elemnél jelezhetjük az al-kép indexet a listában, ez a kép (a kis vagy a nagy verzió) a lista elemmel kapcsolatban lesz használva. A 11.16. ábrán láthatunk egy példát az Items tulajdonság szerkesztőjének használatáról, azokkal az értékekkel, amelyeket a RefList példában használtunk, könyvek, magazinok, CD-ROM-ok és hasonló források listája.



11.16. ábra *A ListView komponens Items tulajdonságának szerkesztője*

Mivel a listához hozzáadtunk néhány al-elemet, ezért a címkéjüket, és az őket megjelenítő oszlopok méretét is be kell állítanunk. A ListView Columns tulajdonságának van erre egy külön szerkesztője. Amikor a ListView komponens tulajdonságaink megadását befejeztük, akkor megírhatjuk a kódot, amellyel válthatunk a különböző nézetek között, és végrehajthatunk más alapvető műveleteket a komponensen. A részleteket megnézhetjük a CD-n található RefList példában, de az Items és Columns tulajdonságok szerkesztési idő alatti beállítása, és a futási idő alatt a kép listák megfelelő hozzárendelése szükséges ahhoz, hogy működő ListView-kat hozzunk létre. A 11.17. ábrán láthatunk néhány példát a ListView kimenetére.



11.17. ábra *A ListView komponens eltérő ViewStyle tulajdonságokkal*

A RefList példában könnyedén megváltoztathatjuk a ListView komponens stílusát a View menüben. Ebben a menüben négy rádiógomb menü elem van, mindegyik a komponens ViewStyle tulajdonságának egy lehetséges értékét állítja be. Íme az OnClick esemény kezelők közül az egyik: (a másik három nagyon hasonló):

```
procedure TForm1.LargeIcons1Click(Sender: TObject);
begin
    ListView1.ViewStyle := vsIcon;
    LargeIcons1.Checked := True;
end;
```

Grafikus rácsok

A Delphi másik érdekes grafikai komponenscsoportját a Gridek (rácsok) alkotják. A rendszer többféle rács komponenst tartalmaz: szövegrácsokat, egy képeket tartalmazó, adatbázissal kapcsolatos rácsokat, és a színrácsokat. Az első két fajta rács különösen hasznos, mert velük sok információt lehet megjeleníteni, és a felhasználó mozoghat bennük. Természetesen a rácsok nagyon fontosak az adatbázis-programozásnál, ahogy azt a 16. és 17. fejezetben látni fogjuk, melyek az adatbázis és kliens-szerver programozással foglalkozik.

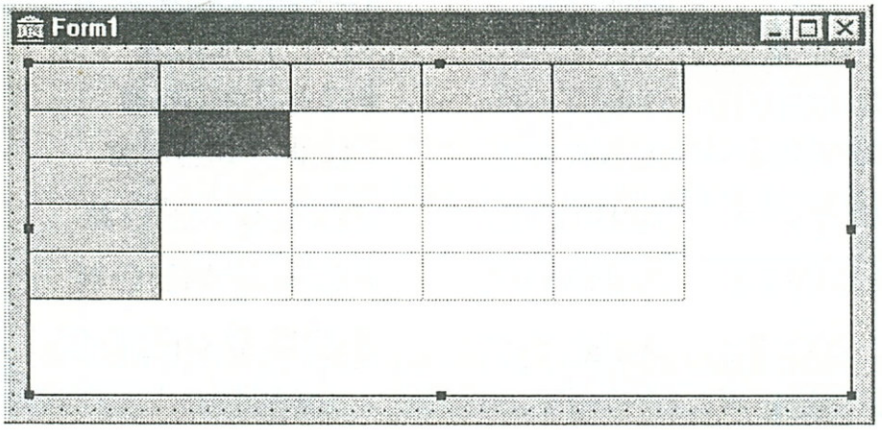
A DrawGrid és a StringGrid komponensek közeli kapcsolatban vannak. Valójában a TStringGrid a TDrawGrid alosztálya. Mi a feladata ezeknek a rácsoknak? Alapvetően adatokat tárolhatunk bennük, akár szövegeket a StringGriddel kapcsolatban, vagy más adatstruktúrákat, és azokat megjeleníthetjük adott feltételekkel. Amíg a szövegrácsokat majdnem úgy használhatjuk, ahogy vannak (mivel a szerkesztési funkciót biztosítják), az általános objektumok rácsai általában több kódolást igényelnek.

A rácsok valójában megjelenítésre szolgálnak, nem pedig tárolásra. Az egyetlen rács, amelyik tárolja is az adatot, amelyet megjelenít az a StringGrid (szövegrács). Az összes többi rács csak megjelenítő, nem tároló. Egy rács alapvetően adott számú oszlopból, és sorból épül fel, amely jelzi a nem görgethető rácsterületet (ahogy azt a 11.18. ábrán is láthatjuk). A rácsok a legösszetettebb komponensek a Delphiben, amit jelez a tulajdonságaik és metódusaik nagy száma is.

A felhasználói felületet illetően a rácsnak lehetnek különböző méretű és számú sorai. Beállíthatjuk minden egyes oszlop vagy sor méretét egymástól függetlenül, mert a RowSize, ColWidth, és a RowHeight tulajdonságok tömbben vannak tárolva. A felhasználói műveleteket illetően, a felhasználó átméretezheti az oszlopokat, és a sorokat (goColSizing és goRowSizing), megragadva átmozgathat egész oszlopokat, és sorokat egy új pozícióba (goRowMoving és goColumnMoving), választhat automatikus szerkesztést, és

intervallumokat jelölhet ki. Mivel sokféle műveletet végezhet a felhasználó a rácsokon, ezért sokféle eseménye is van, mint például az OnColumnMoved, OnDrawCell, vagy az OnSetEditText.

A legfontosabb esemény valószínűleg az OnDrawCell. Erre az eseményre válaszolva a programnak a rács egy bizonyos celláját meg kell rajzolnia. Csak a szövegrácsok tudják automatikusan megjeleníteni a tartalmukat. A DrawGrid valójában nem biztosít adattárolási lehetőséget: egyszerűen csak egy eszköz, amellyel a képernyő egy részén rendezett formában jeleníthetünk meg információkat. Az olyan metódusokat, mint a CellRect, amely visszaadja a cella területét határoló keretet, vagy a MouseToCell, amely visszaad egy meghatározott helyen lévő cellát, öröm használni, figyelembe véve, hogy kezelnek átméretezhető sorokat és oszlopokat, illetve görgethető rácsokat is.



11.18. ábra *Amikor egy új rács komponenst helyezünk a formra, akkor alapértelmezés szerint egy rögzített sort, és oszlopot tartalmaz*

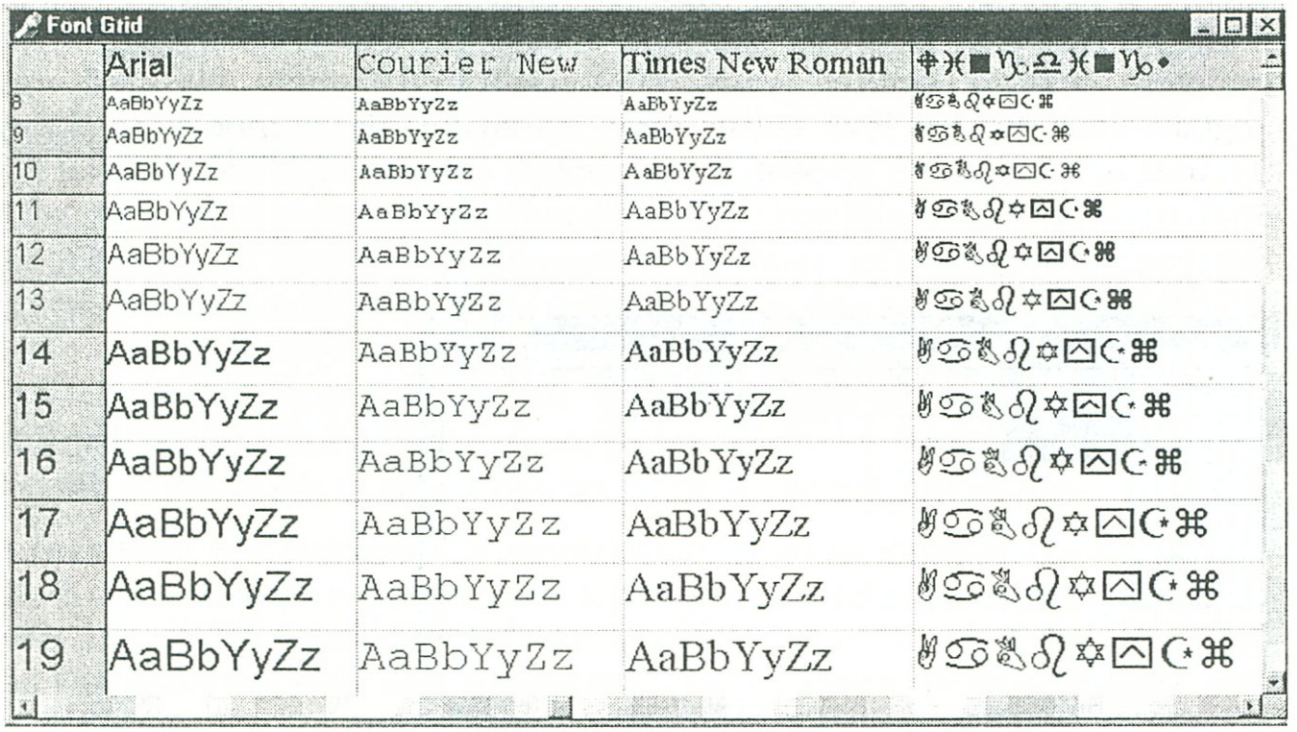
Mire használhatunk egy rácsot? Egy táblázatkezelő létrehozása lehet az első dolog ami az eszünkbe juthat, de Delphiben már van egy OCX komponens táblázatkezelő képességgel. A két rácsot fogjuk használni két példa felépítésére, amelyeket a következő két részben mutatunk be. A StringGrid-et abban a példában használjuk, amely rendszerben installált betűkészletek listáját jeleníti meg, a DrawGrid-et pedig egy aknakereső játékban. Az aknakereső játék utánzata után jön a ColorGrid használatának bemutatása.

Megjegyzés

A Delphi jól használható játékok írására is. Ezeknek valójában sokkal nagyobb a rendszer-, és fejlesztőeszköz igényük, mint bármilyen más kategóriájú szoftvernek, tehát ha megfelel a játékírásra, az azt jelenti, hogy ez egy nagyon hatékonyan fejlesztő környezet.

A font rács

Ha egy StringGrid komponenst teszünk a formra , és az opcióit megfelelően beállítjuk, akkor lesz egy teljes, működő, rácsba rendezett szövegszerkesztőnk, amely nem igényel programozói munkát. Hogy a példát érdekesebbé tegyük, a rács minden celláját különböző fonttal rajzoljuk ki, melyek mérete és típusa is különböző. A FontGrid program eredményét láthatjuk a 11.19. ábrán.



11.19. ábra *A FontGrid alkalmazás kimenetének példája*

A program formja nagyon egyszerű. Csak egy Grid komponenst kell a formra tenni, a felhasználói felülethez kell igazítani, be kell állítani néhány tulajdonságot és opciót, és a program elvégzi a többit. Az oszlopok, és a sorok száma, és a mérete valójában futási idő alatt van kiszámolva. A fontos tulajdonságok, amelyeket meg kell adnunk a DefaultDrawing, amit hamisra kell állítanunk, hogy úgy rajzoljuk ki a rácsot, ahogy nekünk tetszik, és az Options:

```
object Form1: TForm1
  Caption = 'Font Grid'
  OnCreate = FormCreate
  object StringGrid1: TStringGrid
    Align = alClient
    ColCount = 20
    DefaultColWidth = 200
    DefaultDrawing = False
    RowCount = 20
```

```
      Options = [goFixedVertLine, goFixedHorzLine,
        goVertLine, goHorzLine, goDrawFocusSelected,
        goColSizing, goColMoving, goEditing]
      OnDrawCell = StringGrid1DrawCell
    end
  end
```

Ahogy az általában történik a Delphiben, minél egyszerűbb a form, annál bonyolultabb a kód. Ez a példa is követi ezt a „szabályt", habár csak két metódusa van: egy, amelyik inicializálja a rácsot indításkor, és egy, amelyik kiírja az elemeket. A szerkesztés valójában nincs átírva és a rendszerfontot használja. Az első a két metódusból a FormCreate. Induláskor, ez a metódus egy globális Screen objektumot használ a rendszerbe installált betűtípusok elérésére.

Megjegyzés

A Screen globális változó egy TScreen típusú objektumot tartalmaz, amely viszont sok információt tartalmaz a képernyő attributumairól, a rendszerbe telepített betűtípusokról, és az alkalmazás formjairól. A 25. fejezetben lesz részletesen szó a Screen objektumról.

A rácsnak van egy oszlopa minden egyes betűtípushoz, és egy rögzített oszlopa, amelyben a fontok méretei láthatók. Az oszlopok neve a Screen objektumból van átmásolva az oszlopok első sorába (amelynek nulla az indexe):

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I, J: Integer;
begin
  StringGrid1.ColCount := Screen.Fonts.Count + 1;
  StringGrid1.ColWidths [0] := 50;
  for I := 1 to Screen.Fonts.Count do
  begin
    {write the name of the font in the first row}
    StringGrid1.Cells [I, 0] :=
      Screen.Fonts.Strings [I-1];
    {compute maximum required size of column,
    getting the width of the text with the
    biggest size of the font in that column}
    StringGrid1.Canvas.Font.Name :=
      StringGrid1.Cells [I, 0];
    StringGrid1.Canvas.Font.Size := 32;
    StringGrid1.ColWidths [I] :=
      StringGrid1.Canvas.TextWidth ('AaBbYyZz');
  end;
```

A fenti kód utolsó részében a program kiszámolja az összes oszlop szélességét, kiértékelve az oszlopban lévő fonttal, és a program által használt legnagyobb mérettel (32) írt „AaBbYyZz" string által foglalt helyet (az első sorba írva, cellák [I,0]). Hogy kiszámoljuk a szöveg által foglalt helyet, alkalmazzuk a TextWidth és TextHeight metódusokat a Canvason, a megfelelő fontot kiválasztva.

A sorok száma mindig 26, és a növekvő magasságukat a 15 + I x 2 megközelítő képlettel számoljuk ki. Valójában a legmagasabb szöveg kiszámítása azt jelenti, hogy ellenőrizzük a szöveget minden oszlopban, de ez valóban bonyolult lenne ehhez a példához. A megközelítéses képlet elég jól működik, ahogy azt a 11.19. ábrán is láthatjuk, ha futtatjuk a programot. A sorok első cellájában a program kiírja a font méretét, amely megfelel a sor száma plusz hétnek.

Az utolsó művelet a „AaBbYyZz" string tárolása a rács nem rögzített celláiban. Ezt egymásba ágyazott ciklussal értük el. Amikor rácsokkal dolgozunk gyakran használunk majd egymásba ágyazott ciklusokat. Itt a FormCreate metódus második része:

```
    StringGrid1.RowCount := 26;
    for I := 1 to 25 do
    begin
      {write the number in the first column}
      StringGrid1.Cells [0, I] := IntToStr (I+7);
      {set an increasing height for the rows}
      StringGrid1.RowHeights [I] := 15 + I*2;
      {insert default text in each column}
      for J := 1 to StringGrid1.ColCount do
        StringGrid1.Cells [J, I] := 'AaBbYyZz'
    end;
    StringGrid1.RowHeights [0] := 25;
end;
```

Most már tanulmányozhatjuk a második metódust, a StringGrid1DrawCellt, amely megfelel a rács saját OnDrawCell eseményének:

```
procedure TForm1.StringGrid1DrawCell(Sender: TObject;
  Col, Row: Longint; Rect: TRect; State:
TGridDrawState);
```

Ennek a metódusnak van néhány paramétere:

- Col és Row arra cellára hivatkoznak, amellyel éppen dolgozunk.
- Rect a cella területe, amelyet kitöltünk.
- State a cella állapotát írja le, amely három kapcsolóból (flag) áll, amelyek egyszerre is működhetnek: gdSelected (a cella ki van választva), gdFocused (a cellán áll az input fókusz), és a gdFixed (a cella a rögzített területen van, amelynek általában különböző háttérszíne van). A cella állapotának ismerete fontos, mert gyakran hatással van a kimenetére.

A DrawCell metódus a rács megfelelő elemének a szövegét írja ki, az oszlop által használt fonttal, és a sor által jelzett mérettel. Íme a metódus forráslistája:

```
procedure TForm1.StringGrid1DrawCell(
  Sender: TObject; Col, Row: Integer;
  Rect: TRect; State: TGridDrawState);
begin
  {select a font, depending on the column}
  if Col = 0 then
    StringGrid1.Canvas.Font.Name := 'Arial'
  else
    StringGrid1.Canvas.Font.Name :=
      StringGrid1.Cells [Col, 0];

  {select the size of the font, depending on the row}
  if Row = 0 then
    StringGrid1.Canvas.Font.Size := 14
  else
    StringGrid1.Canvas.Font.Size := Row + 7;

  {select the background color}
  if gdSelected in State then
    StringGrid1.Canvas.Brush.Color := clHighlight
  else if gdFixed in State then
    StringGrid1.Canvas.Brush.Color := clBtnFace
  else
    StringGrid1.Canvas.Brush.Color := clWindow;

  {output the text}
  StringGrid1.Canvas.TextRect (
    Rect, Rect.Left, Rect.Top,
    StringGrid1.Cells [Col, Row]);

  {draw the focus}
  if gdFocused in State then
    StringGrid1.Canvas.DrawFocusRect (Rect);
end;
```

A font nevét az oszlop első sorából (0. Sor) nyerjük ki. A font méretét a sor számához hetet adva számoljuk ki (a rögzített oszlopok alapértelmezés szerinti értékeket használnak ehelyett). Beállítva a betűtípust és a méretét, a program kiválaszt egy háttérszínt a cellának a lehetséges állapotától függően: kiválasztott, rögzített, vagy normál (azaz nincs különleges stílusa).

A stílus gdFocused kapcsolójának értékét néhány sorral később használjuk a tipikus fókusz keret megrajzolására – egy téglalap vékony pontozott vonallal. Amikor minden be van állítva, akkor a program végre kirajzolhatja a szöveget, és esetenként fókuszkeretet rajzolhat a fenti StringGrid1DrawCell metódussal.

Tipp

Ha a rács cellájába akarunk szöveget írni, a Canvas TectRect metódusát használtuk a sokkal általánosabb TextOut metódus helyett. Ennek az oka az, hogy a TextRect vágja a szöveget az adott területen, megakadályozva a kereten kívülre írást. Ez különösen fontos a rácsoknál, mivel a cella tartalmának nem szabad kilógnia a határain túlra. Mivel mi az egész rács felszínére rajzolunk, amikor egy cellát rajzolunk, könnyedén megváltoztathatjuk a szomszédos cellák tartalmát is.

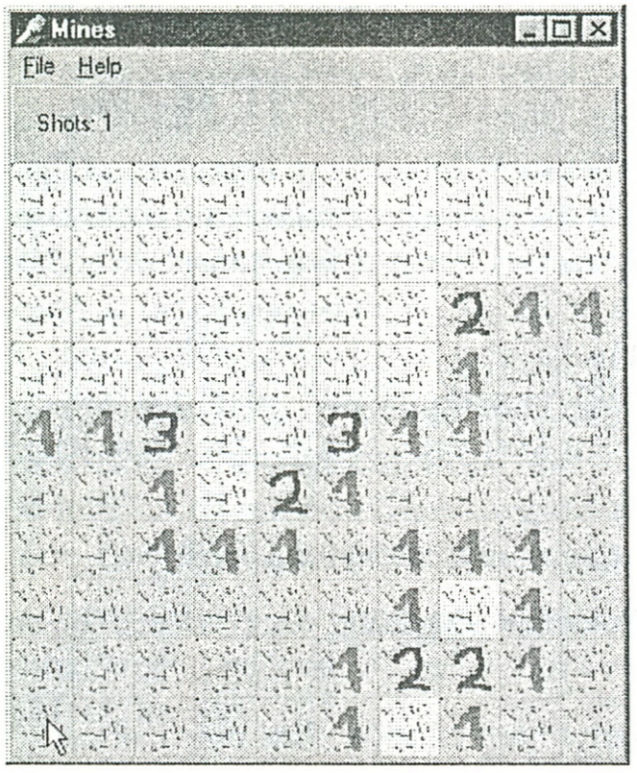
Utolsó megjegyzésként, ne felejtsük el, hogy amikor kiírjuk a rács egy cellájának a tartalmát, nem csak az alapértelmezés szerinti képet kell megjelenítenünk, hanem különböző kimenetet kell biztosítanunk a kiválasztott celláknak, megfelelően rajzoljuk meg a fókuszt, és így tovább. Az ODList példában, amelyben az owner-draw kódot mutattuk, kihagyhattuk volna a ezt a kódot, mert a Delphi kezeli ezeket a speciális eseteket.

Aknák a rácsban

A StringGrid komponens a Cells tömböt használja az elemek értékének tárolására, és van egy Objects tulajdonsága a cellák szokásos értékének tárolására. A DrawGrid komponensnek nincsen előre definiált tárolója. Ezért a következő példában definiálni fogunk egy kétdimenziós tömböt a rács celláinak értékének tárolására – ami a játéktér.

A Mines példa a MineSweeper (aknaszedő) Windowsos játék hasonmása. Ha még sohasem játszottunk ezzel a játékkal, akkor javaslom, hogy próbáljuk ki, mert itt csak alapvető leírást közlünk. Amikor a program elindul, akkor egy üres mezőt (rácsot) jelenít meg, amelyben aknák vannak elrejtve. A bal egér gombbot megnyomva egy cellán tesztelhetjük, hogy van-e akna azon a pozíción. Ha találunk egy aknát, akkor az felrobban, és a játéknak vége. Vesztettünk.

Ha nincs ott akna, akkor a program jelzi a cellában, a körülötte lévő nyolc cellában lévő aknák számát. A cellához közel lévő aknák számának ismeretében jó tippünk lehet a következő körre. További segítség számunkra az, hogy ha egy cella körül nincs akna, akkor a körülötte található cellákban is automatikusan kijelzi a környezetükben található aknák számát, és ha azok közül is van olyan, amelyik körül nincs akna, akkor a folyamat megismétlődik. Ha szerencsések vagyunk, akkor egyetlen kattintással felfedhetünk elég sok aknát (11.20. ábra).



11.20. ábra *A Mines program egy szerencsés kattintás után. Cellák egész csoportja jelent meg egyszerre, amelyekben nem volt akna*

Ha úgy gondoljuk, hogy aknát találtunk, akkor kattintsunk rá a cellára a jobb egér gombbal, elhelyezve ott egy zászlót. A program nem mondja meg, hogy a következtetésünk helyes volt-e, a zászló csak egy tipp a következő kísérleteinkhez. Ha később meggondolnánk magunkat, akkor ismét a jobb egér gombbal a cellára kattintva visszavonhatjuk a zászlót. Ha az összes aknára zászlót tettünk, akkor nyertünk, és vége a játéknak.

Ezek a játékszabályok. Most meg kell valósítanunk őket, egy DrawGridet használva kiindulópontnak. Ebben a példában a rács rögzített, és nem átméretezhető, vagy módosítható futási idő alatt. Valójában 30×30 pixeles négyzet alakú cellákból áll, amelybe ugyanekkora bitmapeket jelenítünk meg.

A program kódja bonyolult, és nem könnyű kiinduló pontot találni, ahonnan elkezdhetjük magyarázni. Ezért a mellékelt CD-n található forráskódban sokkal több megjegyzés van , hogy végigböngészve a kódot megérthessük, hogy mit csinál. Ennek ellenére a legfontosabb elemeket leírom, kiemelve a kulcsfontosságú metódusokat és a használt algoritmusokat, és megpróbáljuk követni a logikai menetét. Ha tudjuk, akkor nyomtassuk ki a példa forráskódját, ha nem, akkor pedig nézzük a monitoron, miközben olvassuk a leírást.

Először is, a program adatai két tömbben vannak tárolva (a form privát mezőiként deklarálva):

```
Display: array [0 .. NItems - 1, 0 .. NItems -1] of
Boolean;
Map: array [0 .. NItems - 1, 0 .. NItems -1] of Char;
```

Az első egy Boolean értékekből álló tömb, amely jelzi, hogy egy elem megjeleníthető-e, vagy maradjon rejtve. Figyeljük meg, hogy a tömb sorainak, és oszlopainak száma NItems. Szabadon változtathatjuk ezt a konstanst, de akkor ennek megfelelően át kell méreteznünk a rácsot. A második tömb a Map, amely az aknák, és a zászlók pozícióját tárolja, és a környező aknák számát. A tömb karakter kódokat használ a megfelelő megszámlálható adattípus helyett, a 0-8 számjegyek előnyös alkalmazhatóak a környező aknák számának jelzésére. Itt van a kódok listája:

- M: Mine, jelzi a játékos által még meg nem talált akna pozícióját
- K: Known mine, jelzi a játékos által már megtalált, és zászlóval jelzett akna pozícióját
- W: Wrong mine, a játékos olyan helyre tett zászlót, ahol nincs akna
- 0-8: a környező cellákban található aknák száma

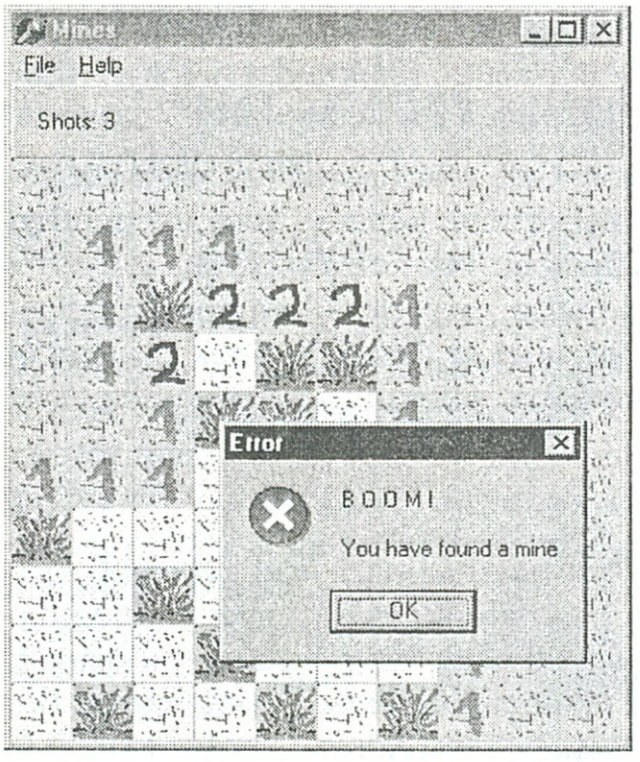
Az első metódus, amelyet átnézünk, a FormCreate lesz, amely az induláskor fut le. Ez a metódus inicializálja a form osztály mezőit, feltölti a két tömböt alapértelmezés szerinti értékekkel (két egymásba ágyazott ciklussal), és a rácsban elhelyez aknákat. Egy konstans által megadott alkalommal (ami az aknák száma) a program véletlenszerű pozícióba rak egy aknát. Ha ott már van egy akna, akkor a ciklusnak újra le kell futnia, hogy az aknák száma a Map tömbben megegyezzen a megadottal. Ha nem így lenne, akkor a programnak nem lenne soha vége, mivel ellenőrzi, hogy a talált aknák száma megegyezik-e a rácsban elhelyezett aknák számával. A következő a ciklus kódja, amely többször is lefuthat mint NMines-szer, a MinesToPlace egész változónak köszönhetően, amelyet megnövelünk, amikor egy már létező aknára helyezünk egy másikat:

```
Randomize;

  {place 'NMines' non-overlapping mines}
MinesToPlace := NMines;
while MinesToPlace < 0 do
begin
  X := Random (NItems);
  Y := Random (NItems);
  if Map [X, Y] <> 'M' then     {if there isn't a mine}
  begin
    Map [X, Y] := 'M';      {add a mine}
    Dec (MinesToPlace)
  end;
end;
```

Az inicializáló kód utolsó része számolja ki a környező aknák számát azokban a cellákban, ahol nincs akna. Ezt a ComputeMines eljárás cellánkénti meghívásával érjük el. A függvény kódja meglehetősen bonyolult, mivel figyelembe kell vennie azokat a speciális eseteket, amikor a rács határához közel vannak az aknák. Az eljárás meghívásának eredményét, a környező cellákban található aknák számát reprezentáló karaktert, a Map tömbben tároljuk.

A következő logikai eljárás a DrawGrid1MouseUp, amely meghívódik, ha az egérgombot felengedjük egy cella fölött (volt egy kis probléma a fókusszal a mouse-down eseményben, ezért választottuk a mouse-up eseményt). A metódus először kiszámítja, hogy melyik cellán kattintottunk az egérrel, a rács MouseToCell metódusának meghívásával. Ezután három különböző kódrészlet van: egy rövid, amikor a játék befejeződik, és a másik kettő a két egér gombra.

Amikor a bal egér gombot megnyomjuk, a program ellenőrzi, hogy van-e ott akna (rejtett vagy sem), és ha van akkor befejezi a programot egy robbanással (11.21. ábra).



11.21. ábra *Hoppá! Aknára léptünk!*

Ha nincs ott akna, a program a cella Display értékét igazra állítja, és ha ott nulla van, akkor elindítja a FloodZeros eljárást. Ez a metódus jeleníti meg a látható cellához – amelynek az értéke nulla – közeli nyolc elemet, ismételve a műveletet újra és újra, ha azok között is van nulla értékű cella. Ez a rekurzív hívás bonyolult, mert biztosítani kell egy módot a befejezésére. Ha van két cella

közel egymáshoz, mindkettőnek nulla az értéke, mindegyik a másik közelében van, ezért örökké kérni fogják egymást, hogy jelenítsék meg magukat, és a körülöttük lévő cellákat. Ismétlem, a kód bonyolult, és áttanulmányozásának legjobb módja a hibakeresőben való nyomon követés.

Ha a jobb egér gomb volt megnyomva, akkor a program megváltoztatja a cella állapotát M-ről (rejtett akna) K-ra (ismert akna) és oda-vissza (abban az esetben, ha ott tényleg van akna), vagy egy számról W-re (rossz akna) és oda-vissza (ha ott nincs akna). A jobb gomb feladata a képernyőn a zászló kapcsolgatása, a felhasználó bármikor visszavonhat egy lerakott zászlót, ha úgy gondolja, hogy a korábbi döntése rossz volt. Ha az összes aknát megtalálták, a program egy gratuláló üzenettel befejeződik.

Egy nagyon fontos kódrészlet az OnMouseUp eseményre válaszoló metódus végén van. Minden alkalommal, ha a felhasználó rákattint egy cellára, és annak a tartalma megváltozik, akkor a cellát újra kell rajzolni. Ha az egész rácsot újra kellene rajzolni, akkor a program lassabb lenne. Ezért a Windows API InvalidateRect funkcióját használtuk, amelyet már a 10. fejezetben bemutatott Shapes4 példában láthattunk:

```
MyRect := DrawGrid1.CellRect (Col, Row);
InvalidateRect (DrawGrid1.Handle, @MyRect, False);
```

Az utolsó fontos metódus a DrawGrid1DrawCell. Már használtuk ezt a megjelenítő eljárást az előző példában, tehát emlékeznünk kellene, hogy ezt minden újrarajzolandó cellánál meg kell hívni. Alapjában véve ez a metódus kinyeri a cellának megfelelő kódot, amely megjeleníti a megfelelő bitmapet, amelyet egy fájlból töltünk be. Készítettünk egy bitmapet az összes képnek, ezt egy új erőforrás-fájlba tároljuk, amelyet a form kódjába csatolunk a programhoz ezzel a direktívával:

```
{$R BITMAPS.RES}
```

Emlékezzünk rá, hogy amikor erőforrásokat használunk a kód gyorsabb szokott lenni, mint amikor különálló fájlokat használunk, és így egyetlen futtatható fájlt kapunk, amit terjeszthetünk. A bitmapek neve megfelel a rács kódjának, „M" karakterrel az elején, mivel a „0" név érvénytelen lenne. A bitmapeket betölthetjük és kirajzolhatjuk a cellába a következő kóddal:

```
StrPCopy (Name, 'M' + Code);
Bmp.Handle := LoadBitmap (HInstance, Name);
DrawGrid1.Canvas.Draw (Rect.Left, Rect.Top, Bmp);
```

Természetesen ez csak akkor fut le, ha a cella látható, azaz ha a Display igaz. Máskülönben egy alapértelmezés szerinti bitmap jelenítődik meg (a bitmap neve „UNDEF"). A bitmapek betöltése az erőforrás fájlból minden alkalommal lassúnak tűnik , ezért a program betölthetné a bitmapeket a memóriába, ahogy azt a World2 példában is tettük korábban a fejezetben. Azonban most egy

másik, kevésbe hatékony megközelítést választunk, a Cache-t. Ennek azért van értelme, mert már használunk erőforrásokat fájlok helyett a dolgok felgyorsítására.

A Mines bitmap cache kicsi, mivel csak egy eleme van, de megléte meglehetősen felgyorsítja a programot. A program tárolja az utoljára használt bitmapet, és a kódját; ezután minden alkalommal, amikor új elemet kell kirajzolnia, és a kód ugyanaz, akkor a cache-ben lévő bitmapet használja. Itt a fenti kód új verziója:

```
if not (Code = LastBmp) then
begin
  StrPCopy (Name, 'M' + Code);
  Bmp.Handle := LoadBitmap (HInstance, Name);
  LastBmp := Code;
end;
DrawGrid1.Canvas.Draw (Rect.Left, Rect.Top, Bmp);
```

A cache méretének növelése kétségkívül javítani fog a sebességen. Tekinthetjük a bitmapek listáját egy nagy cache-nek, de ez valószínűleg haszontalan, mivel néhány bitmapet (a nagy számokkal rendelkezők) ritkán használunk. Ahogy láthatjuk, érdemes egy kicsit fejleszteni a program felgyorsítása érdekében, és sokat tehetünk a felhasználói felület fejlesztése érdekében is. Ha megértettük a programnak ezt a verzióját, akkor képesek leszünk jelentős mértékben fejleszteni ezt.

Színek választása

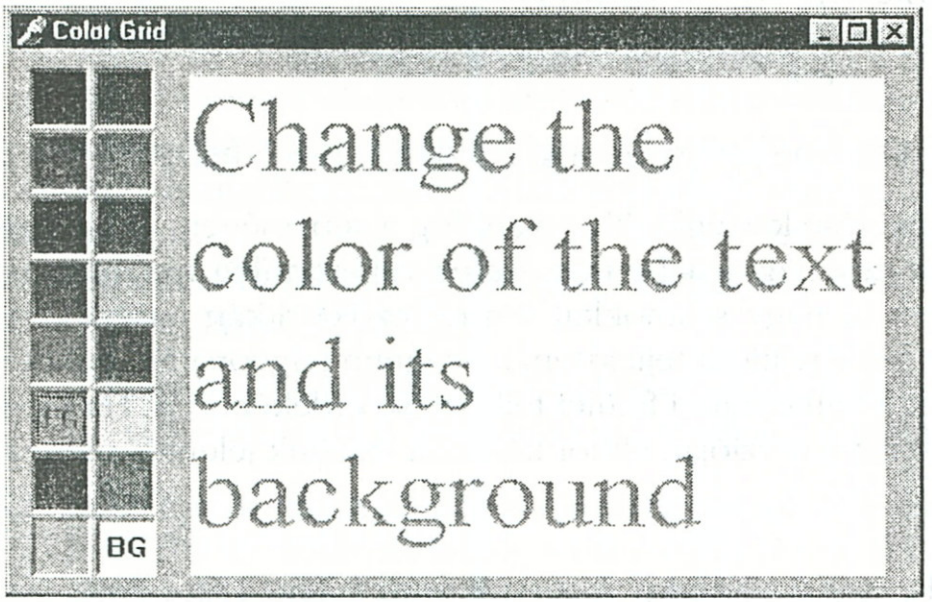
Az utolsó példában a rács használata sokkal egyszerűbb, mint az előzőekben. A komponens paletta Samples oldalán lévő ColorGrid komponenst használjuk. Ennek a komponensnek nincs sok köze más rácsokhoz, és használata egyszerű. Jól használható minden alkalommal, ha a felhasználó egy-két színt kiválaszthat (előtér és háttér, alapvetően) az általános Color párbeszédablak használata nélkül. A ColorGrid különösen hasznos, amikor azt akarjuk, hogy a felhasználó válasszon ki egy színt, és rögtön mutatunk egy előzetest az eredményről. A Delphi maga is használja ezt a kontrolt az Environment Options párbeszédablak **Colors** oldalán, amelyen be lehet állítani a szintaxis kiemelést.

A ColorGridnek van néhány cellája, mindegyikben egy színnel, és használja a bal, és a jobb egérgomb kattintást két szín kiválasztására, név szerint az előtér színéhez a bal egérgombot, a háttérszínhez a jobb egérgombot. Két tulajdonság, a ForegroundIndex, és a BackgroundIndex tartalmazzák az aktuális színek értékét, amelyeket a felhasználó automatikusan megváltoztat kattintással, de csak akkor, ha aktívak.

Az egyetlen különös tulajdonság a GridOrdering, amely jelzi, a rács struktúráját, azaz a sorok és oszlopok számát. Mivel a ColorGridben mindig 16 szín van, kiválaszthatunk egy függőleges, vagy vízszintes sort, vagy két sorban 8 színt,

vagy egy 4 × 4-es négyzetrácsot. Készítettünk egy példát, ColGrid, amiben van egy rács, és egy szövegcímke, amely a rácsban feltüntetett színeket használja (11.22. ábra). A forráskód csak két sorból áll. Minden alkalommal, ha a felhasználó kattint a rácson, a két szín átmásolódik a címkére:

```
procedure TForm1.ColorGrid1Change(Sender: TObject);
begin
  Label1.Color := ColorGrid1.BackgroundColor;
  Label1.Font.Color := ColorGrid1.ForegroundColor;
end;
```



11.22. ábra *A ColorGrid komponensre épülő ColGrid példa*

A TeeChart használata

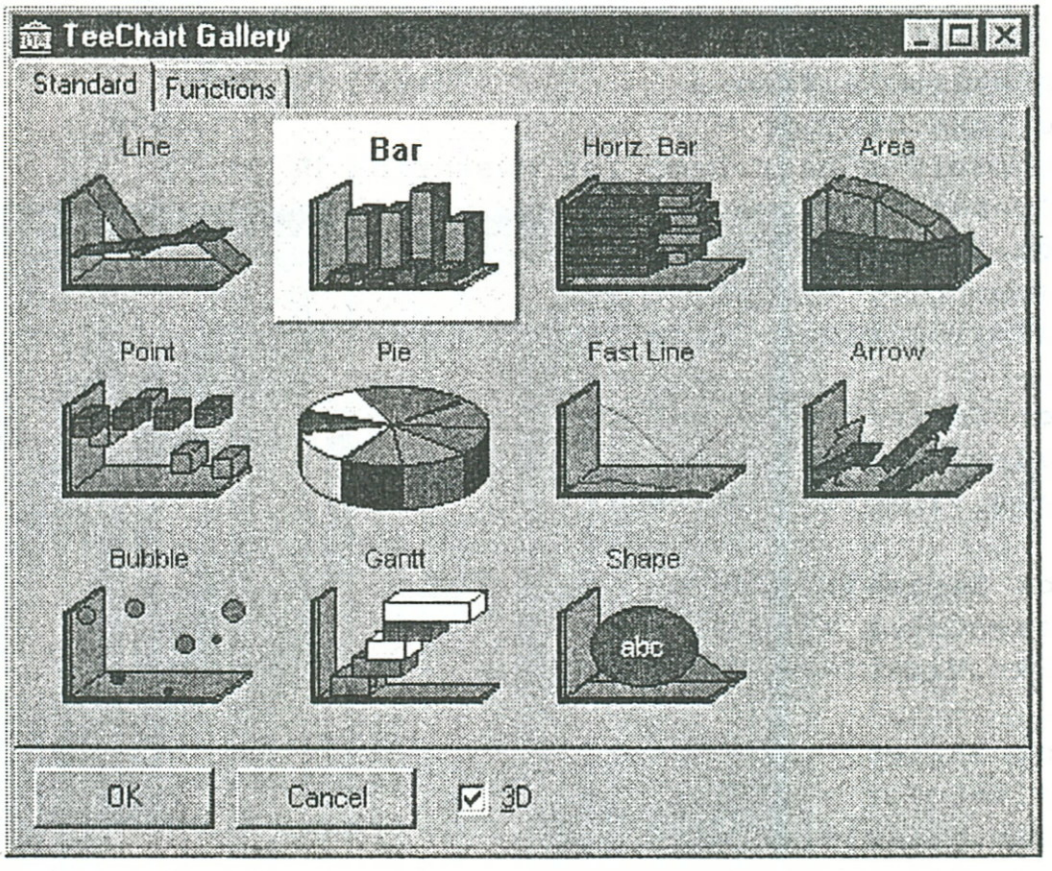
Az utolsó komponens, amelyet megvizsgálunk ebben a fejezetben a TeeChart komponens, egy VCL alapú grafikonkezelő komponens, amelyet David Berneda készített a Borland cég Delphi 3 Developer és Client/Server verziója számára. A Delphi előző verziói tartalmaztak üzleti grafikus kontrolokat, de azok OCX kontrolok voltak, nem a Delphi sajátja. Azonfelül a TeeChart egy rendkívül jó komponens, nem véletlenül került bele a csomagba.

A TeeChart komponens nagyon összetett: A Delphiben van egy Súgó fájl, és más dokumentáció ehhez a komponenshez, ezért nem fogunk időt szakítani összes képességének leírására. Csak néhány példát készítünk, és majd a 16. fejezetben újra használjuk egy adatbázis-kezelő példában. A TeeChartnak valójában háromféle verziója van: önálló komponens (a komponens paletta Additonal oldalán), adatkezelő verzió (a Data Controls oldalon), és egy Report verzió (jelentés készítő, a QuickReport oldalon).

Megjegyzés

Természetesen egyszerűbb lenne a TeeChart varázslóval létrehozni egy példát, de látva az összes lépést, jobban megérthetjük a komponens struktúráját.

A TeeChart komponens biztosítja az alapvető struktúrát a táblázatkezeléshez, valamint támogatja a tervezést és a vizuális tárolást táblázatok részére. Az aktuális táblázatok a TChartSeries osztály objektumai, vagy származtatott osztályok. Ha egy formra teszünk egy TeeChart komponenst, akkor egy, vagy több sorozatot létre kell hozzá hoznunk. Ezt a Chart Component Editor-t megnyitva tehetjük meg: válasszuk ki a komponenst, kattintsunk a jobb egér gombbal, hogy megjelenjen a form tervező lokális menüje, és válasszuk ki az **Edit Chart** parancsot. Most nyomjuk meg az **Add** gombot, és válasszuk ki a grafikont (vagy a sorozatot – series), amit hozzá akarunk adni a sok választható közül (ahogy azt a 11.23. ábrán is láthatjuk).



11.23. ábra *A TeeChart Gallery-ben kiválaszthatjuk a táblázat és a sorozat típusát*

Amikor új sorozatot hozunk létre, egy új TChartSeries alosztály objektum adódik a formhoz. Ugyanúgy működik, mint a MainMenu komponens, amely a formhoz TMenuItem osztályú objektumokat ad hozzá (vagy a Table komponens, amely TField alosztályú objektumokat ad hozzá, amit majd a 16. fejezetben látni fogunk). A TSeries objektumot a Chart Component Editor-ban vagy az Object Inspectorban szerkeszthetjük

A különböző TChartSeries alosztályoknak – a különböző fajtájú grafikonok – különböző tulajdonságaik, és metódusaik vannak (ámbár vannak közösek is). Ne felejtsük el, hogy egy grafikonnak lehetnek különböző sorozatai is: ha mind ugyanolyan típusú, akkor valószínűleg jobban illeszkednek, mint a többféle típusúak esetében. Egyébként lehet több összetett, különböző típusú, egyszerre látható grafikonunk. Időnként ez nagyon jól használható lehetőség.

Az első példa készítése

Egy TeeChart komponenst helyeztünk a formra, és utána hozzáadtunk négy 3D Bar sorozatot, amelyek a TBarSeries osztály objektumai. Ezután beállítottunk néhány globális tulajdonságot, mint például a táblázat címe, és így tovább. Itt van az összegzése ezeknek az információknak, a form szöveges leírásából véve:

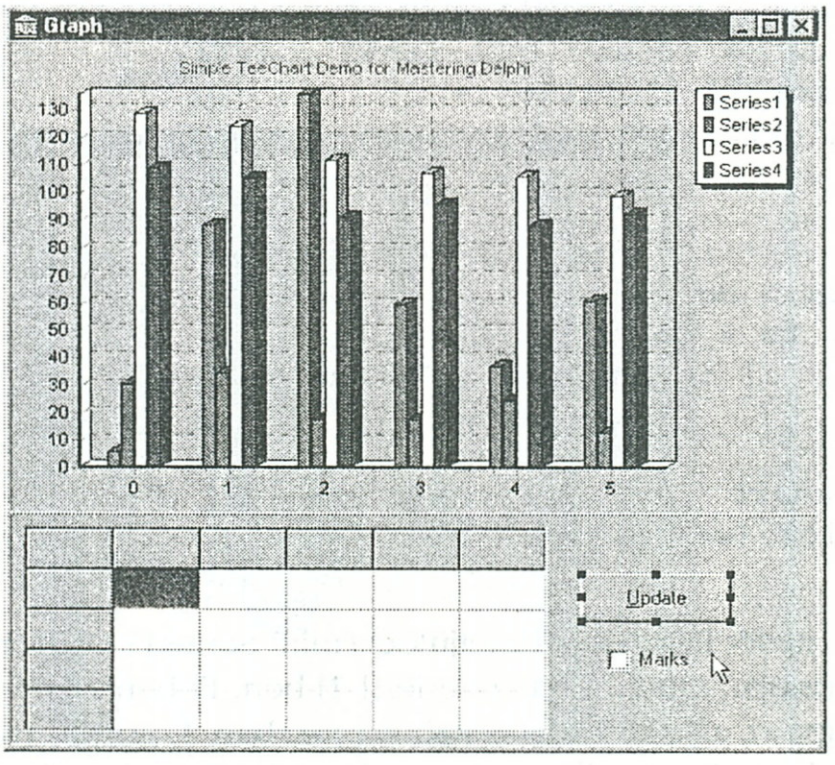
```
object Chart1: TChart
  AnimatedZoom = True
  Title.Text.Strings = (
    'Simple TeeChart Demo for Mastering Delphi')
  BevelOuter = bvLowered
  object Series1: TBarSeries
    SeriesColor = clRed
    Marks.Visible = False
  end
  object Series2: TBarSeries
    SeriesColor = clGreen
    Marks.Visible = False
  end
  object Series3: TBarSeries
    SeriesColor = clYellow
    Marks.Visible = False
  end
  object Series4: TBarSeries
    SeriesColor = clBlue
    Marks.Visible = False
  end
end
```

Ezután, a formhoz hozzáadtunk egy szövegrácsot, és egy nyomógombot, Update címkével. Ezt a gombot használjuk a rács numerikus értékeinek a táblázatba való másolására. A rács egy 5 x 4 alapú mátrix, plusz egy sor, és oszlop a címeknek. Az ide tartozó rész a következő:

```
object StringGrid1: TStringGrid
  ColCount = 6
  DefaultColWidth = 50
  Options = [goFixedVertLine, goFixedHorzLine,
    goVertLine, goHorzLine, goEditing]
  ScrollBars = ssNone
  OnGetEditMask = StringGrid1GetEditMask
end
```

A RowCount tulajdonságnak alapértelmezés szerinti értéke 5, és nem jelenik meg a szöveges leírásban (ugyanez jellemző a FixedCols és FixedRows tulajdonság 1-es értékére is). Egy szövegrács egy fontos eleme a szűrő (Edit Mask), amelyet az összes cellája használ. Ezt az OnGetEditMask eseményt használva állíthatjuk be:

```
procedure TForm1.StringGrid1GetEditMask(Sender: TObject;
  ACol, ARow: Longint; var Value: string);
begin
  // edit mask for the grid cells
  Value := '09;0';
end;
```



11.24. ábra *A TeeChart komponensen alapuló Graph1 példa szerkesztési idő alatt*

Valójában van még egy komponens, egy jelölőnégyzet, amellyel be-, és kikapcsolhatjuk a sorozatok jeleinek láthatóságát (a jelek kis sárga címkék, amelyek leírnak minden értéket). A 11.24. ábrán szerkesztés alatt láthatjuk a formot. Ebben az esetben a sorozatok véletlenszerű értékkel vannak feltöltve, ez egy hasznos szolgáltatása a komponensnek, mert így meg tudjuk nézni a kimenetét egyetlen valódi adat beírása nélkül.

Adatok hozzáadása a grafikonhoz

Most egyszerűen csak inicializálnunk kell a szövegrács adatait, és a grafikon sorozataiba másolnunk azt. Ez a form OnCreate eseményének kezelőjében kap helyet. Az OnCreate metódus kitölti a rács rögzített elemeit és a sorozatneveket, és ezután tölti fel a szövegrács adatterületét, és végül meghívja az Update gomb OnClick eseményét, hogy frissítse a grafikont:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I, J: Integer;
begin
  with StringGrid1 do
  begin
    {fills the fixed column and row,
    and the chart series names}
    for I := 1 to 5 do
      Cells [I, 0] := Format ('Group%d', [I]);
    for J := 1 to 4 do
    begin
      Cells [0, J] := Format ('Series%d', [J]);
      Chart1.Series [J-1].Title :=
        Format ('Series%d', [J]);
    end;

    Randomize;     {fills the grid with random values}
    for I := 1 to 5 do
      for J := 1 to 4 do
        Cells [I, J] := IntToStr (Random (100));
  end; // with

  // update the chart
  UpdateButtonClick (self);
end;
```

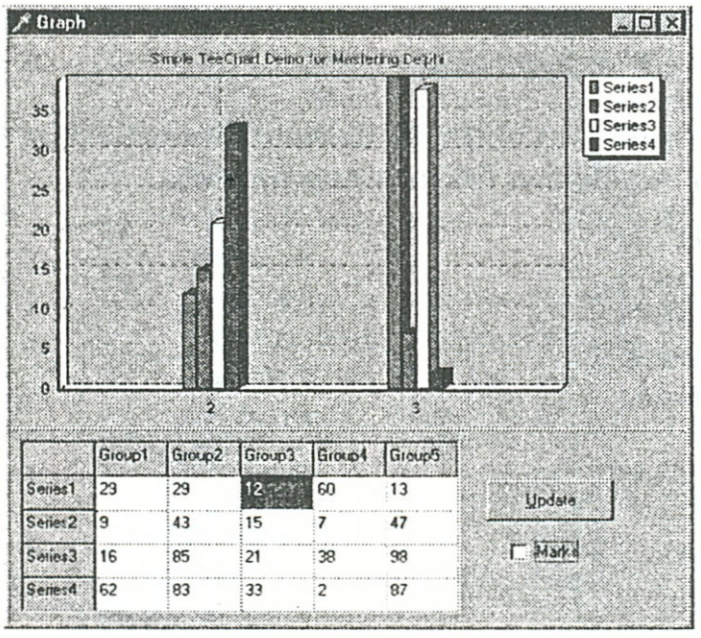
A sorozatokat a komponens nevét használva érhetjük el (mint Series1), vagy a táblázat Series tömb tulajdonságát, mint a Chart1.Series[J-1]-ben. Ebben a kifejezésben figyeljük meg, hogy az aktuális adat a szöveg rácsban 1. sortól, és oszloptól indul, az első sor és oszlop (nullás index-szel) a rögzített elemeket tartalmazzák – míg a táblázat sorozatai nullától indulnak.

Egy másik példa a sorozatok frissítésére a jelölőnégyzet OnClick esemény kezelőjében van jelen, amely kapcsolgatja a jelek láthatóságát:

```
procedure TForm1.ChBoxMarksClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 4 do
    Chart1.Series [I-1].Marks.Visible :=
      ChBoxMarks.Checked;
end;
```

A legérdekesebb kód azonban az UpdateButtonClick metódusban van, amely frissíti a táblázatot. A program ezt úgy oldja meg, hogy először kitörli a meglévő adatokat mindegyik táblázatból, és utána hozzáadja az új adatokat (vagy adatpontokat, hogy zsargont használjunk):

```
procedure TForm1.UpdateButtonClick(Sender: TObject);
var
  I, J: Integer;
begin
  for I := 1 to 4 do
  begin
    Chart1.Series [I-1].Clear;
    for J := 1 to 5 do
      Chart1.Series [I-1].Add (
        StrToInt (StringGrid1.Cells [J, I]),
        '', Chart1.Series [I-1].SeriesColor);
  end;
end;
```



11.25. ábra *A Graph1 példa formja futási idő alatt. Figyeljük meg, hogy a grafikont kinagyítottuk*

Az Add metódus paraméterei (amit akkor használunk, ha nem akarunk megadni X értéket, csak egy Y értéket) az aktuális érték, a címke, és a szín. Ebben a példában a címkét nem használjuk, egyszerűen csak kihagyjuk. Színnek használhattuk volna az alapértelmezés értéket, a clTeeColor-t, hogy megkapjuk a sorozatok megfelelő színét. Ha akarjuk, akkor használhatunk saját színeket az adatok különböző intervallumainak jelzésére.

Ha kész van a grafikon, akkor a TeeChart sokféle megjelenítési lehetőséget kínál fel számunkra. Könnyedén nagyíthatjuk (a bal egérgombbal jelezve a területet), kicsinyíthetünk (az egeret ellenkezőleg használva, a bal felső sarokba húzva), a jobb egérgombbal pedig mozgatni lehet a nézetet. A 11.25. ábrán láthatunk példát a nagyításra.

Sorozatok dinamikus létrehozása

A Graph1 példa bemutat néhányat a TeeChart komponens képességeiből, de csak egy rögzített típusú grafikonon alapul. Fejleszthettük volna azzal, hogy a függőleges oszlopok alakját változtathatóvá tesszük, de ehelyett egy sokkal általánosabb módszerrel foglalkozunk, mégpedig lehetővé tesszük a felhasználó számára, hogy válasszon különböző sorozatokból és grafikonokból.

A TeeChart komponensnek eleinte ugyanazok lesznek a tulajdonságai mint az előző példában. A form viszont most már négy kombinált listát tartalmaz, egyet a rács minden sorához. Minden kombinált lista négy értéket tartalmaz (Line, Bar, Area, és Point) megfelelően az általunk kezelni kívánt sorozatok típusának. Valójában, hogy rugalmasabban kezelhessük ezeket a kombinált listákat a kódban, a formhoz privát mezőihez hozzáadtunk egy tömböt, amely ezeket a kontrolokat tartalmazza:

```
private
  Combos: array [0..3] of TComboBox;
```

Ez a tömb a FormCreate metódusban fel van töltve az aktuális komponensekkel, amely elemeik kezdő elemét is beállítja. A FormCreate új kódja:

```
// fill the Combos array
Combos [0] := ComboBox1;
Combos [1] := ComboBox2;
Combos [2] := ComboBox3;
Combos [3] := ComboBox4;

// show the initial chart type
for I := 0 to 3 do
  Combos [I].ItemIndex := 1;
```

Megjegyzés

Ez a példa bemutatja, hogyan lehet általában a Delphiben létrehozni kontrolok tömbjét, amire a Visual Basic programozók már régóta vártak. A Delphi annyira rugalmas, hogy a kontrolokból álló tömb nem beépített: úgy hozhatjuk létre őket ahogy akarjuk. Ez a fajta megközelítés azon a tényen alapul, hogy általánosságban hozzárendelhetünk egy eseménykezelőt különböző eseményekhez is, amelyet a VB nem enged meg.

Az összes kombinált lista ugyanazt az OnClick esemény kezelőt használja, ami a táblázat jelenlegi sorozatait megsemmisíti, és újakat hoz létre, majd ezután frissíti azok tulajdonságait és adatait:

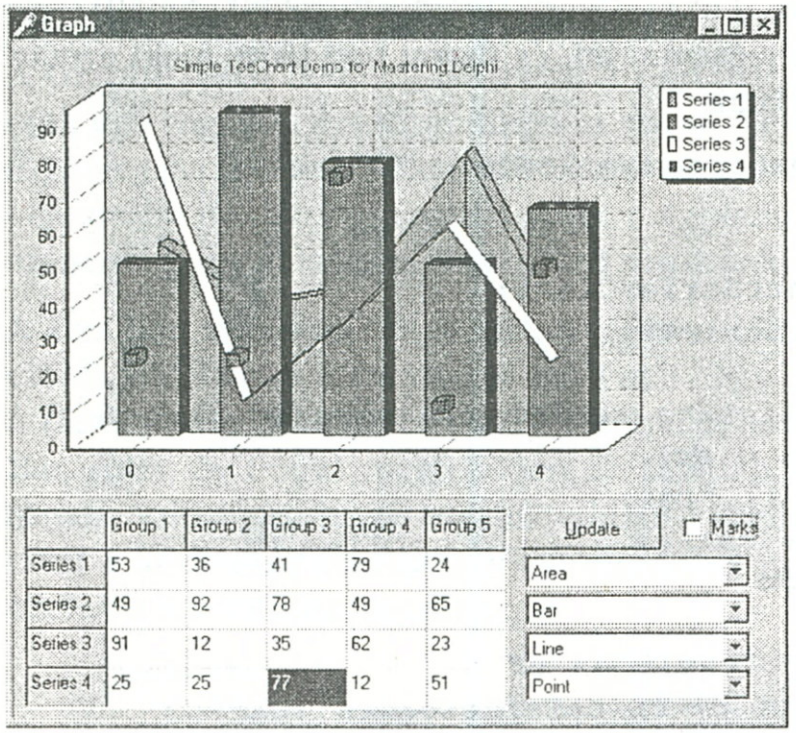
```
procedure TForm1.ComboChange(Sender: TObject);
var
  I: Integer;
  SeriesClass: TChartSeriesClass;
  NewSeries: TChartSeries;
begin
  // destroy the existing series (in reverse order)
  for I := 3 downto 0 do
    Chart1.Series [I].Free;
  // create the new series
  for I := 0 to 3 do
  begin
    case Combos [I].ItemIndex of
      0: SeriesClass := TLineSeries;
      1: SeriesClass := TBarSeries;
      2: SeriesClass := TAreaSeries;
    else // 3: and default
      SeriesClass := TPointSeries;
    end;
    NewSeries := SeriesClass.Create (self);
    NewSeries.ParentChart := Chart1;
    NewSeries.Title :=
      Format ('Series %d', [I + 1]);
  end;
  // update the marks and update the data
  ChBoxMarksClick (self);
  UpdateButtonClick (self);
end;
```

A kód központi része a `case` kifejezés, amely egy új osztályt tárol a SeriesClass osztály hivatkozás változóban, amellyel új sorozat objektumokat hozhatunk létre, és beállíthatjuk vele a ParentChart-ot és a Title-t. Használhattuk volna a táblázat AddSeries metódusát mindegyik case ágban, és utána állítottuk volna be a Title-t egy másik ciklussal. Valójában egy ilyen hívás:

```
Chart1.AddSeries (TBarSeries.Create (Self));
```

létrehoz egy sorozat objektumot, és egyben beállítja a szülő táblázatot (parent chart).

A program új verziójában minden sorozat grafikonját egymástól függetlenül megváltoztathatjuk. A végeredményt a 11.26. ábrán láthatjuk.

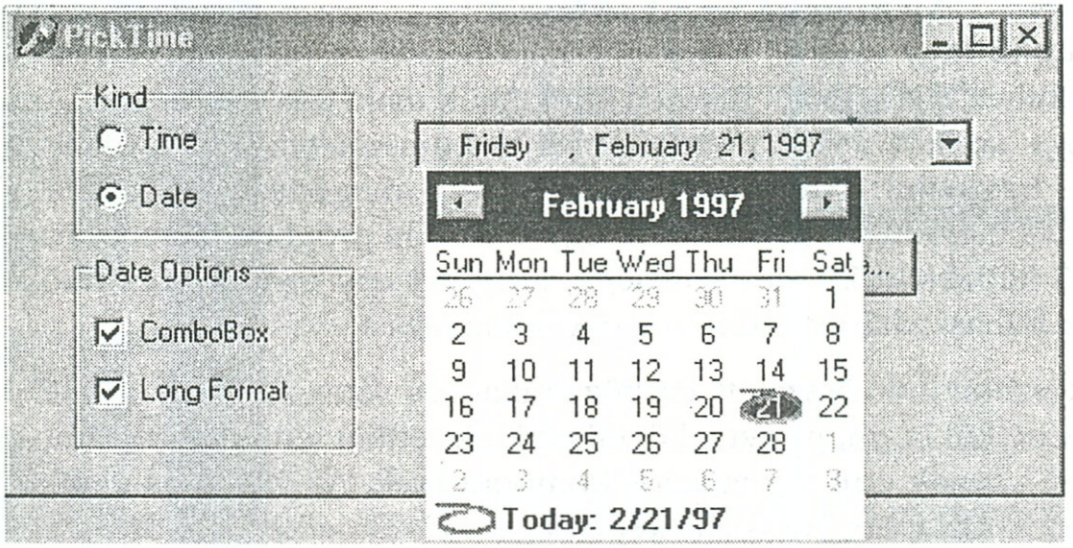


11.26. ábra *Többféle grafikon és táblázat sorozatok, amelyet a Graph2 példa jelenít meg*

A Dátum–Idő kiválasztó kontrol

Van még egy grafikus komponens, amelyről röviden beszélünk még a fejezetben, a DateTimePicker kontrol. Ez egy új kontrol, amelyet mostanában adott hozzá a Microsoft a Windows 95 általános vezérlő könyvtárához (és a Microsoft Internet Explorerrel terjesztette). Ne felejtsük el, hogy a felhasználóknak meg kell lennie a könyvtár frissített verziójának, hogy használhassák ezt a kontrolt.

A DateTimePicker egy egyszerű dátum-, és időpontbeállító eszköz. Szerkeszthetjük benne az időt (alapértelmezés szerint a pontos időt) növelve a részeit (óra, perc, másodperc), vagy a kurzorbillentyűket, vagy a kis fel-le gombokat használva. A dátum kiválasztására van még egy lehetőség: használjuk a listát, amiben megjelenik egy legördülő naptár, amilyet a 11.27. ábrán is láthatunk.



11.27. ábra *Az egyszerű PickDate példa kimenete, a legördülő naptárral. Új dátum beírására is használhatjuk a komponenst*

Az ábra egy nagyon egyszerű példa kimenete, melynek PickDate a neve, és e köré a komponens köré épült, hogy tesztelhessük annak kulcsfunkcióit. Két rádió gombbal, és két jelölőnégyzettel kapcsolgathatjuk a Kind tulajdonság értékét (amely lehet dtkTime vagy dtkDate), és a DateFormat tulajdonság értékét (vagy dfLong, vagy dfShort). Ez a két utolsó tulajdonságnak csak akkor van jelentősége, ha a kontrol a dátumot mutatja.

A mellékelt CD-n megnézhetjük a példa forráskódját. Ahogy látni fogjuk, van benne még két gomb, amelyek egy párbeszédablakban jelennek meg, és a kiválasztott időt és dátumot jelzik. A DateTimePicker kontrol Date, és Time tulajdonságainak (egy TDate, és egy TTime érték) értékét kiolvastuk, és a DateToStr, és a TimeToStr Delphi függvényekkel szöveggé alakítva valósítottuk meg ezt. Már használtunk néhányat ezek közül a fügvények közül a 4. fejezetben a TDateTime adat típusnál. Íme az egyik gomb OnClick esemény kezelőjének az egyszerű kódja:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MessageDlg ('The time is ' +
    TimeToStr (DateTimePicker1.Time),
    mtInformation, [mbOK], 0);
end;
```

Hogyan tovább?

Ebben a fejezetben felfedeztünk több különböző Delphi komponenst, beleértve az Outline-okat, rácsokat, bitmap gombokat, és néhány Windows 95-ös általános kontrolt, mint a TreeView, és a ListView. Természetesen vannak más, hasonló típusú komponensek is, beleértve a komponenspaletta Samples oldalán találhatókat.

Létre is hozhatunk vagy vásárolhatunk új komponenseket, melyeket hozzáadhatunk a Delphihez. Ezt a témát röviden áttekintjük majd a könyv végén. Ami fontos az, hogy amikor a komponensek jól dokumentáltak, akkor nagyon könnyű őket használni. Nincs kell semmit sem megérteni eltekintve a tulajdonságok és metódusok jelentésétől. Egy jó Delphi programozónak biztos alapokra és jó képzelőerőre van szüksége, amellyel egy formon különböző komponensek felhasználásával létrehoz egy alkalmazást.

Egyszerűen készíthetünk összetett eszköztárakat, és státuszsorokat a létező komponensek nélkül is, ahogy azt a következő fejezetben tenni fogjuk, ámbár a Delphi 3-ban vannak erre használatos komponensek. Az általános grafikus komponensek csak alapvető rajzolási képességekkel rendelkeznek, de meglehetősen egyszerű kibővíteni a felhasználói felületüket és a működésüket. Újra használni fogunk néhány ebben a fejezetben leírt grafikus komponenst, sokkal összetettebb programok írására.

12. FEJEZET

Eszköztár, CoolBar, státuszsor

E fejezet tartalma:

- Kontrolok csoportosítása paneleken
- Eszközsor készítése panelből
- Tippek a formokon
- Szolgáltatások az eszköztáron
- Windows 95-ös Toolbar (Eszköztár) kontrol
- Egy remek eszköztár

Eszköztár, CoolBar, státuszsor

A Windows alkalmazások egyik legfeltűnőbb jellemzője az ablak tetejénél lévő eszköztár, és a lenti státuszsor. Az eszköztárnak, a program készítőjétől függően több neve is van, mint a SpeedBar, vagy a control bar, irányító panel, ezek némelyike bejegyzett védjegy. De akárhogy is hívjuk, az eszköztár általában kis gombokból áll, amikre kattintva parancsokat adhatunk ki, vagy opciókat kapcsolgathatunk. Néha, az eszköztáron állhat kombi panel, szerkesztőmező, vagy valamilyen más kontrol. Az újabb alkalmazások eszköztárait általában elmozgathatjuk az ablak bal vagy jobb oldalára, vagy akár kis, gombokkal teli lebegő ablak formájában eszköztárként használhatjuk

A Microsoft Internet Explorer újfajta eszköztárat mutatott be, aminek átlátszó, lapos gombjai és eltérő viselkedése volt. Ezt CoolBar-nak nevezték el, és a Delphi 3 tartalmazza a használatához szükséges Windows 95 komponenst, de ez csak egyike a Delphi számos újításának ezen a területen.

Ettől eltérően a státuszsor általában egy, vagy több szöveges részből áll, amik a program állapotát jelölik. Ezen helyet kaphat az egér pozíciója, az éppen használt betűtípus, megjelenhetnek rajta tippek, hibaüzenetek és így tovább, hiszen a státuszsor felépítése csak az alkalmazástól függ.

Kontrolok csoportosítása paneleken

Státuszsor vagy eszköztár elkészítéséhez Delphiben a direkt erre szolgáló komponens mellett használhatjuk a Panel komponenst is, amire tehetünk gombokat, vagy másik paneleket is. Most ezt nézzük meg először, a másik megközelítést csak ezután vizsgáljuk.

Paneleket használhatunk a kliens terület részekre osztására, és a komponensek csoportosítására is. Bár a paneleknek lehet saját szövegük, ezt ritkán használjuk, inkább a kinézetükkel tesznek jó szolgálatot, ilyenkor a panel az elválasztóra (Bevel) hasonlít, de ez utóbbinak sokkal kisebb a jelentősége és a funkcionalitása is, hiszen az elválasztó egy grafikus komponens, és nem lehet más kontrolok szülője.

Grafikai szempontból a panel dupla elválasztóként szerepelhet, mert két ilyen paramétere is van, a BevelInner (belső elválasztó) és BevelOuter (külső elválasztó). Ha ehhez még hozzávesszük a BevelWidth (elválasztó szélesség) lehetséges értékeit, akkor ezzel rengeteg féle hatást elérhetünk, ahogy azt a tulajdonságok állítgatásával, vagy a CD-n lévő PanBord példa futtatásával kipróbálhatjuk. Általában, ha panelt készítünk, akkor azt az Alignment tulajdonság állításával helyezzük el a formon. Eszköztár esetére az alTop (fent), státuszsor imitálásához az alBottom (lent) érték használható, de készíthetünk függőleges paneleket is.

A már megszokott tulajdonságok mellett a Panel komponens képes tippek kezelésére is; ehhez a Hint (tipp) és a ShowHints (tippek mutatása) a két legfontosabb tulajdonság. A tipp kifejezés általában azokat a rövid üzeneteket jelöli, amik akkor jelennek meg, ha az egérmutatót az eszköztár egy gombja fölé visszük, de ugyanez a kifejezés él a státuszsoron megjelenő rövid leírásokra is.

Figyelem!

Zavart okozhat, hogy ugyanazt a kifejezést használjuk két némiképp eltérő dologra, de a Delphi ugyanazt a belső mechanizmust használja a tippek mindkét fajtájára, ezért maradtunk ennél az elnevezésnél. Erről később még lesz szó.

A tippek nem csak a paneleken léteznek. A formoknak is lehetnek tippjeik néhány, vagy akár az összes komponensükhöz, és ebben a fejezetben még fogunk is látni egy ilyen formot.

Eszközsor készítése panelből

Egyszerű eszköztár készítéséhez tegyünk a form kliens területének tetejére egy panelt, és tegyünk rá néhány SpeedButton komponenst. A SpeedButton komponens hasonlít az előző fejezetben ismertetett bitmap gombra. A bitmap gombhoz hasonlóan a speed button-nak is lehet címe (Caption) és képe (Glyph), bár gyakran csak grafikus elemet tartalmaz.

A fő különbség a viselkedésében van. A Bitmap gombok a nyomógombokhoz (Button) hasonlítanak, és általában dialógus ablakokban használjuk őket. A SpeedButton képes nyomógombként, jelölőnégyzetként, vagy rádió gombként viselkedni, és a státuszától függően lehet mindig más képe is. Van még egy technikai különbség is a SpeedButton és a bitmap gomb között. Az első egy grafikus komponens, aminek nincsen ablak leírója, így nem használ Windows erőforrásokat, nem kaphat fókuszt, nincsen tabulátor sorrendje, és gyorsabb elkészíteni és kirajzolni.

Ha kiválasztjuk a SpeedButton komponenst, és egy példányát a panelre – az eszköztárba – tesszük, akkor egy grafikus nyomógombot kapunk. Az eszköztárak gombjai általában kicsik, körülbelül 20 × 20 pixelesek. Ilyen kis helyre nem írhatunk sok szöveget, egy ikon azonban már ekkorában is szemléletes lehet. Azt azonban mindenképpen vegyük figyelembe, hogy a 20 × 20 pixeles ikonok egy kis képernyőn még kisebbek lesznek.

Az első eszköztár

Miután letettük a SpeedButtont, írjunk valamit az OnClick eseményéhez. Ez az alapértelmezett művelete, ezért ha duplán kattintunk rajta tervezés közben, akkor ide jutunk. A Toolbar példában van egy hangjegy ikon, amihez a következő kód tartozik:

```
procedure TToolbarForm.SpeedButtonNoteClick(Sender:
                                                TObject);
begin
  Beep;
end;
```

Most akkor adjunk több SpeedButtont, amik rádiógombokként fognak működni. Tegyünk le néhány gombot, és mindegyiknek állítsuk azonosra a GroupIndex (CsoportIndex) tulajdonságát. Miután mindnek azonos a GroupIndex tulajdonsága, ezek közül a rádió gombokhoz hasonlóan csak egy lehet kiválasztva, de egynek viszont mindenképpen célszerű kiválasztva lennie, ezért még tervezés közben, vagy a program indításakor állítsuk az egyik Down tulajdonságát igazra.

Más esetekben készíthetünk olyan csoportot, ahol szintén legfeljebb gomb egy lehet kiválasztva, de az is lehetséges, hogy egyik sincs – ez olyankor használható, ha a felhasználó kiválaszthat egy opciót, de nem kötelező választania a csoportból. Ilyenkor a felhasználó úgy kapcsolhatja ki az aktuális gombot, ha újra rákattint. Ezt a viselkedést úgy érhetjük el, ha a csoport mindegyik gombján igazra állítjuk az AllowAllUp tulajdonságot. Ebben az esetben természetesen induláskor is lehet mindegyik gomb felengedett.

A két különböző módszer kipróbálására készíthetünk a panelre két SpeedButton csoportot. A Toolbar példában, ami a 12.1. ábrán látható, a címke szövegének igazítására, és a stílusának meghatározására három-három gombot használtunk. Az igazítási gombok közül az egyiknek mindig lenyomva kell lennie, hiszen a szöveg valahova mindig igazítva van. A gombok OnClick eseménykezelői mind a következő felépítést követik:

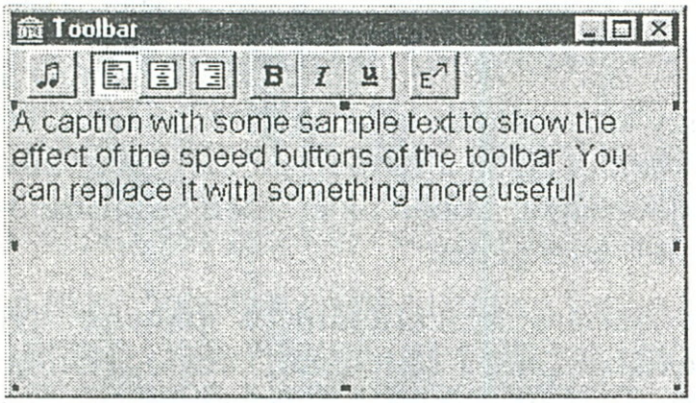
```
procedure TToolbarForm.SpeedButtonLeftClick(Sender:
                                                TObject);
begin
  Label1.Alignment := taLeftJustify;
end;
```

Ettől eltérően a stílus három gombja lehet egyszerre felengedett, hiszen a szöveg stílusa lehet normál. A stílusok ily módon történő kezelése messze áll a tökéletestől, hiszen a stílusok nem zárják ki egymást, használhatunk például „félkövér dőlt" betűket – de most a példa kedvéért ettől eltekintünk. Ezzel a kóddal az a probléma, hogy nem választhatjuk ki egyszerűen a stílust a gomb lenyomásakor, a gomb felengedésekor a stílust vissza kell állítani normálra.

Ezeknek a gomboknak a kattintás esemény kezelőibe a következőhöz hasonlót kell írnunk:

```
procedure TToolbarForm.SpeedButtonBoldClick(Sender:
                                                TObject);
begin
  if SpeedButtonBold.Down then
    Label1.Font.Style := [fsBold]
  else
    Label1.Font.Style := [];
end;
```

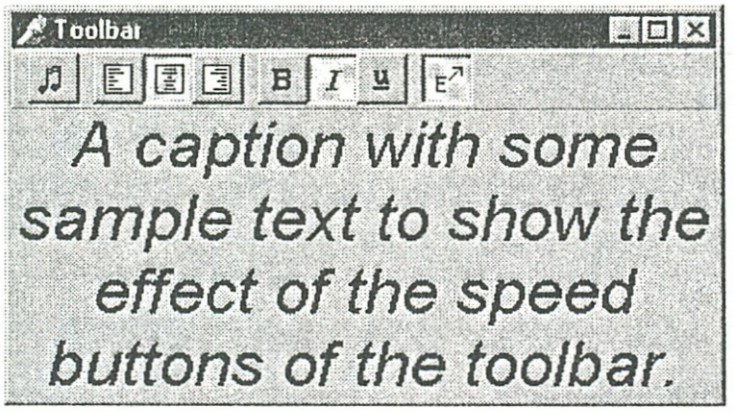
Ehelyett a második csoport helyett használhattunk volna három különálló gombot is, amik jelölőnégyzetekként működtek volna, hiszen a három tulajdonság nem zárja ki egymást. De hogyan készítünk olyan SpeedButtont, ami jelölőnégyzetként működik? Van valami különleges tulajdonság? Ha végignézzük a SpeedButton komponens tulajdonságait, akkor nem találunk semmi ilyesmit. A megoldás azonban már ismert: készítsünk egy olyan gombcsoportot, ami csak egyetlen gombból áll, és engedjük meg, hogy a csoport összes gombja – igen, mind az egy – egyszerre kikapcsolt lehessen.



12.1. ábra *A Toolbar példa formja*

Ez úgy történik, hogy leteszünk egy SpeedButtont, a GroupIndex tulajdonságát egy eddig még nem használt értékre állítjuk, és az AllowAllUp tulajdonságát igazra módosítjuk. Az ennek megfelelő utolsó SpeedButton kódja egyszerű:

```
procedure TToolbarForm.SpeedButtonBigClick(Sender:
                                                TObject);
begin
  if SpeedButtonBig.Down then
    Label1.Font.Size := 24
  else
    Label1.Font.Size := 12;
end;
```



12.2. ábra *A Toolbar példa utolsó gombja jelölőnégyzetként viselkedik*

Ha lenyomjuk ezt a gombot, akkor az lenn is marad, ahogy az a 12.2. ábrán látható; ha újra megnyomjuk, akkor kikapcsolódik. Próbáljuk ki a példát, hogy megértsük a különböző gombok és gombcsoportok működését. A példa összefoglalásaként következzen a form szöveges leírása:

```
object ToolbarForm: TToolbarForm
  Caption = 'Toolbar'
  object Label1: TLabel
    Align = alClient
    AutoSize = False
    Caption = 'A caption ...'
    WordWrap = True
  end
  object Panel1: TPanel
    Align = alTop
    object SpeedButtonNote: TSpeedButton
      Glyph.Data = {note.bmp}
      OnClick = SpeedButtonNoteClick
    end
    object SpeedButtonLeft: TSpeedButton
      GroupIndex = 1
      Down = True
      OnClick = SpeedButtonLeftClick
    end
    object SpeedButtonCenter: TSpeedButton...
    object SpeedButtonRight: TSpeedButton...
    object SpeedButtonBold: TSpeedButton
      AllowAllUp = True
      GroupIndex = 2
      OnClick = SpeedButtonBoldClick
    end
    object SpeedButtonItalic: TSpeedButton...
```

```
      object SpeedButtonUnderline: TSpeedButton...
      object SpeedButtonBig: TSpeedButton
        AllowAllUp = True
        GroupIndex = 3
        OnClick = SpeedButtonBigClick
      end
    end
  end
```

Eszköztár gombok ki és bekapcsolása

Ennek az egyszerű példának nagyjából megszokott a viselkedése. Minden gombnak megvan a saját ikonja, ami a ki és bekapcsolt állapotától független. A SpeedButton-okhoz azonban hozzárendelhetünk több ikont, amik a gomb státuszát jelölik, anélkül, hogy nekünk kellene a képet cserélgetnünk. A két megoldás közötti különbséget az előző fejezetbeli Fire példa két változatában láthattuk.

Ennek a lehetőségnek a kipróbálására készítettünk az eszköztár példából egy második változatot, a Toolbar2-t. Ebben a változatban hozzáadtunk a menühöz néhány olyan pontot, amikkel néhány SpeedButton-t kikapcsolhatunk, vagy elrejthetjük az egész eszköztárat. Ez utóbbit szolgálja a következő kódrészlet, ami a Toolbar menü egyik parancsát kezeli:

```
procedure TToolbarForm.Visible1Click(Sender: TObject);
begin
  {hide or display the toolbar, setting the menu item
    check mark}
  Panel1.Visible := not Panel1.Visible;
  Visible1.Checked := not Visible1.Checked;
end;
```

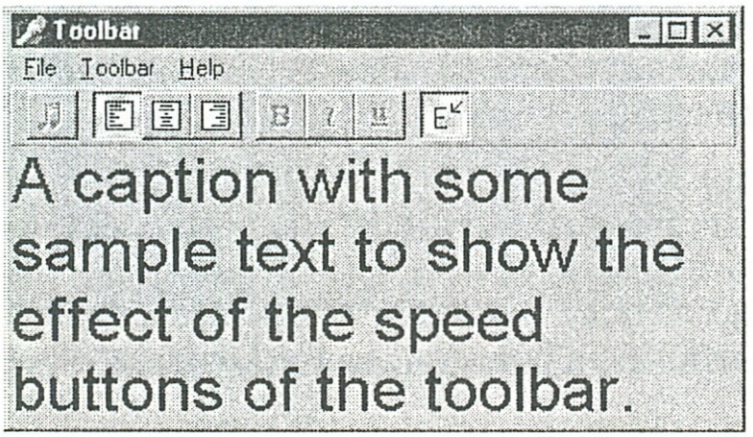
Ez a kód két logikai tulajdonságot kapcsol, az egyiket azért, hogy végrehajtsa a szükséges feladatot, a másikat pedig, hogy hozzáadja, vagy eltávolítsa a menüpont jelölését. Két hasonló kifejezés szerepel a gombok ki és bekapcsolására szolgáló eljárásokban:

```
procedure TToolbarForm.DisableSound1Click(Sender:
                                                TObject);
begin
  {disable or enable button and set menu check mark}
  SpeedButtonNote.Enabled := not
SpeedButtonNote.Enabled;
  DisableSound1.Checked := not DisableSound1.Checked;
end;
```

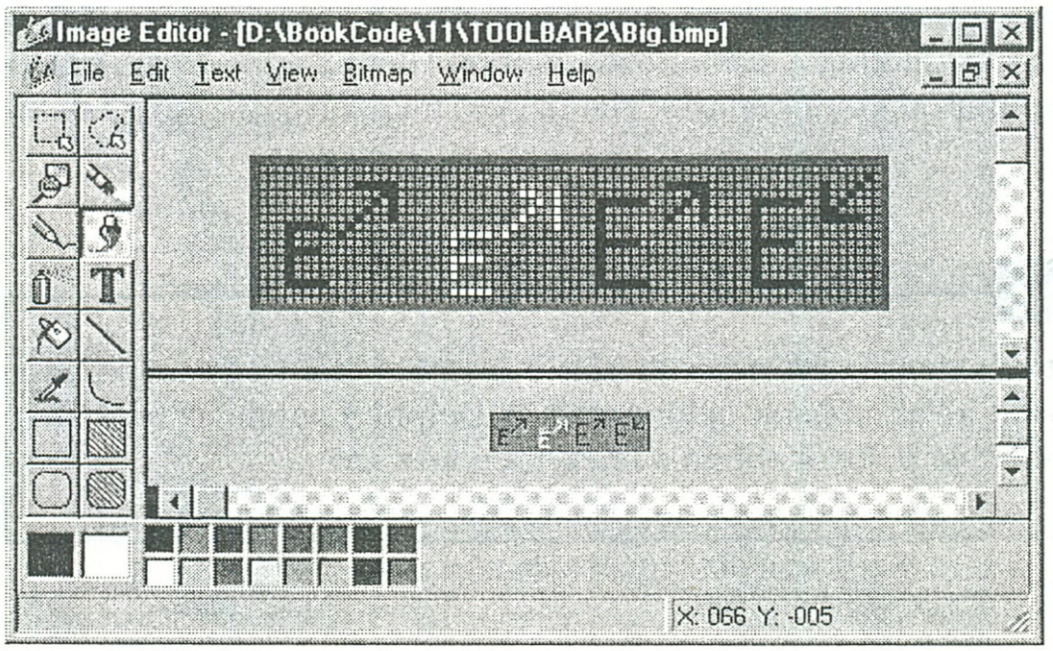
Amikor a gombokat kikapcsoljuk, a Delphi automatikusan megrajzolja a képünk elszürkített változatát – a valóságban a „szürkített" változat színe a

rendszerbeállításoktól függ – ahogy azt megfigyelhetjük, ha összehasonlítjuk a 12.3. ábrát a 12.2. ábrával.

Ebben a példában a Delphi a kép három változatát készíti el: a normálist, a lenyomottat, és a kikapcsoltat. Itt azonban úgy döntöttünk, hogy az utolsó gomb képeit magunk készítjük el, erre szolgál a 12.4. ábrán látható kép, ami négy részből áll. Ehhez a gombhoz ugyan csak két képre van szükségünk, egyre a lenyomott, és egyre a felengedett gombhoz, de adnunk kellett két másikat a különböző állapotoknak, mert a Stay Down (lent marad) ikon a negyedik. Több ikon segítségével animálhattuk a gombot. Ha futtatjuk a programot és lenyomjuk a gombot, akkor a kép azonnal megváltozik és később, a kattintás esemény után újra változik, lásd a 12.3. ábrát.



12.3. ábra *A Toolbar2 példa. Néhány kikapcsolt eszköztár gomb ki van kapcsolva, és az utolsó gomb képe is megváltozott*



12.4. ábra *A Toolbar2 példa utolsó gombján használt bittérkép*

Megjegyzés

A Toolbar példában szürke képeket használtunk, de semmi akadálya nincsen színes képek használatának, sőt ez egyre több helyen kezd elterjedni.

Tippek hozzáadása az eszköztárhoz

Az eszköztárak másik gyakori tartozékai lebegő tippek – olyan szövegek, amik röviden leírják az egérmutató alatti gomb funkcióját. Ez a szöveg általában sárga dobozban jelenik meg, miután az egérmutatót egy ideje a gomb fölött tartottuk. Hogy tippeket adjunk az alkalmazás eszköztárához, egyszerűen állítsuk igazra a panel komponens ShowHints (tippek mutatása) tulajdonságát.

Szintén megváltoztathatjuk az Application (alkalmazás) globális objektum HintColor (tippszín), HintPause (tippszünet), HintHidePause (Tipp elrejtés szünet), és HintShortPause (tipp rövid szünet) tulajdonságait. A Toolbar3 példa lehetővé teszi a felhasználó számára, hogy beállítsa a tippek háttérszínét a Toolbar menü Hint Color (tipp szín) menüpontjával, ami a következő eseménykezelőt tartalmazza:

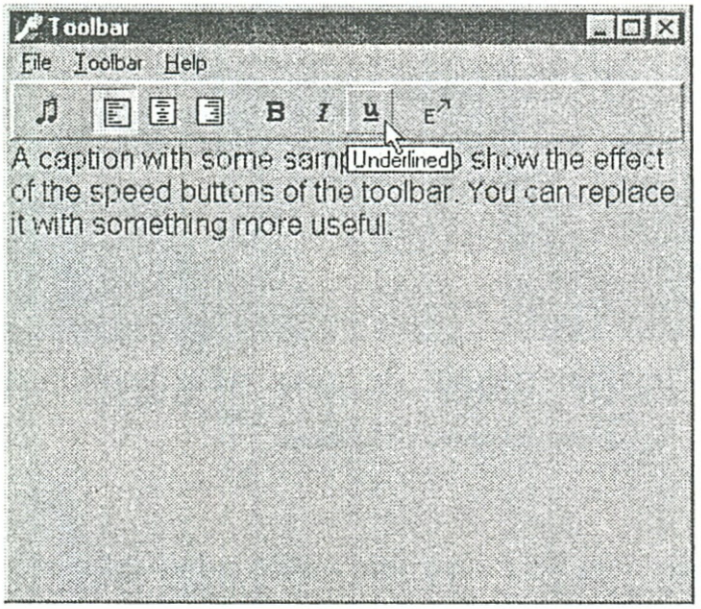
```
procedure TToolbarForm.HintColor1Click(Sender: TObject);
begin
  ColorDialog1.Color := Application.HintColor;
  if ColorDialog1.Execute then
    Application.HintColor := ColorDialog1.Color;
end;
```

Ebben a példában kitöltöttük a különböző gombok Hint tulajdonságait, és beállítottuk a SpeedButton komponensek Flat (lapos) tulajdonságát, amitől új kinézetük lett.

Új!

A Flat tulajdonság azért került bele a Delphi 3-ba, hogy utánozza a Microsoft alkalmazások új kinézetét. A Delphi 3 tartalmaz eszköztár komponenst is, erről ebben a fejezetben lesz szó.

Anélkül, hogy akár egyetlen új Pascal sort írnánk, a felújított programunk jól működik, profinak és modernnek néz ki, ahogy az 12.5. ábrán látszik. Még tehetünk mást is, hogy feljavítsuk a programot: készítünk helyzet érzékeny tippet, és lehetővé tesszük, hogy a felhasználó ki és bekapcsolhassa az új lapos kinézetet.



12.5. ábra *A Toolbar 3 példa lebegő tippekkel, és lapos gombokkal*

A helyzet érzékeny tippnek csak az utolsó gombnál van értelme. A gomb tippje, az „Expand" (növel) csak akkor igaz, ha a gomb nincs lenyomva, máskülönben a tippnek valami másra kellene váltania, mint például „Shrink" (csökkent). Ezt a SpeedButtonBigClick eseményhez hozzáadott két kódsorral végezhetjük el, ami megváltoztatja a gomb tippjét mindig, amikor a státusza megváltozik:

```
procedure TToolbarForm.SpeedButtonBigClick(Sender:
                                                TObject);
begin
  if SpeedButtonBig.Down then
  begin
    Label1.Font.Size := 24;
    SpeedButtonBig.Hint := 'Shrink';
  end
  else
  begin
    Label1.Font.Size := 12;
    SpeedButtonBig.Hint := 'Expand';
  end;
end;
```

A másik változtatás a gombok lapos stílusának kapcsolása, ezt azokért a felhasználókért tesszük, akik nem szeretik az új stílust. A stílus átváltását az összes gombon meg kell tennünk, ehhez a Panel1 komponens – az eszköztár – Controls (kontrolok) tulajdonságát használjuk, ami a panel alá tartozó kontrolok listáját tartalmazza, vagyis most a gombokat. Íme a kód:

```
procedure TToolbarForm.FlatStyle1Click(Sender: TObject);
var
  I: Integer;
begin
  FlatStyle1.Checked := not FlatStyle1.Checked;
  for I := 0 to Panel1.ControlCount - 1 do
    (Panel1.Controls [I] as TSpeedButton).Flat :=
      FlatStyle1.Checked;
end;
```

Nemsokára még tovább bővítjük a Toolbar példát, de most nézzük meg a formokat és a tippeket.

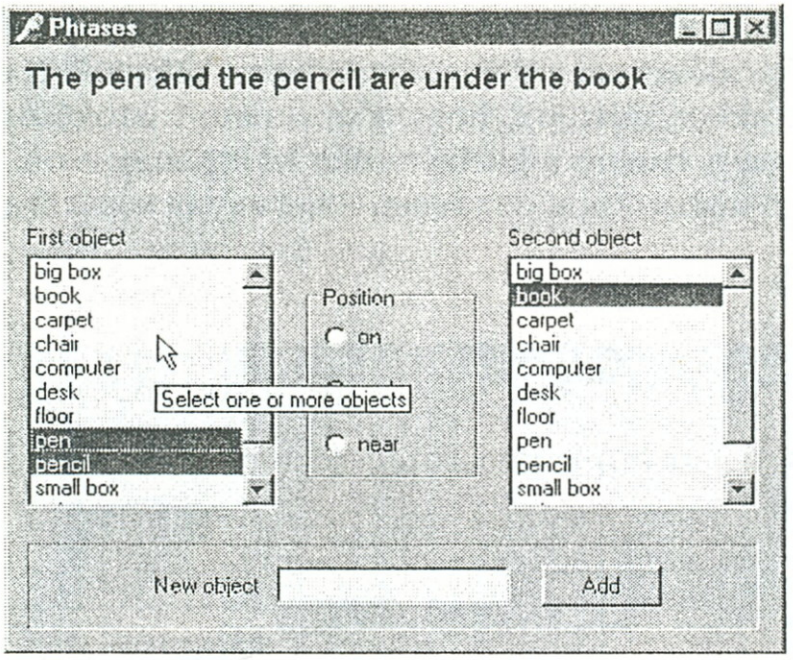
Tippek a formokon

Éppen úgy, ahogy az alkalmazásunk eszköztáránál láthattuk, adhatunk tippeket formokhoz, vagy formok komponenseihez, anélkül, hogy panelre lenne szükségünk. Ezt egy létező programon, a Phrases3 egyszerű példán mutatjuk be. Az új változat, a Pharases4, a 12.6. ábrán látható módon tippeket mutat ahogy az egeret a form komponensei fölé mozgatjuk.

Ahogy az az ábráról, és a program futtatásakor is látszik, a tippek általában a komponensek alatt jelennek meg, ami jó az eszköztár gombjainál, de egy form komponenseinél már nem mindig a legjobb megoldás. Ha a formhoz adunk tippet, akkor az az ablak alatt jelenik meg, a fenti címke tippje pedig, legalábbis kezdetben, jóval a szöveg alatt látszik, hiszen a címke mérete elég nagy, hogy akár három sort is tárolhasson. Az egyetlen módszer, hogy a tippek az egérhez közel jelenjenek meg, és ne az aktuális komponens alatt, a testre szabott tippek alkalmazása, amiről a következő részben lesz szó.

Tippek testre szabása

Láthattuk, hogy milyen könnyen adhatunk tippeket formokhoz és eszköztárakhoz, valamint, hogy a tippek megjelenítését is könnyedén módosíthatjuk. A legegyszerűbb ilyen dolog az Application objektum HintColor tulajdonságának átállítása, ahogy azt a ToolBar3 példában tettük, vagy a három szünetekkel kapcsolatos tulajdonság állítása, ezek a HintPause, HintHidePause, és a HintShortPause. Az első azt adja meg, hogy mennyi ideig kell az egérmutatót a komponens felett tartanunk, mielőtt a tipp megjelenik, a második megadja, hogy a tipp mennyi idő múlva tűnik el, a harmadik pedig azt állítja be, hogy két tipp megjelenítése között mennyit kell várni.



12.6.ábra *A Phrases4 példa. Az első doboz felett hagyott egérmutató hatására megjelenő tipp*

Ha több beleszólást akarunk a megjelenítésébe, akkor tovább módosíthatjuk a tippeket azzal, ha az alkalmazás OnShowHint (tipp megjelenítésekor) eseményéhez egy metódust rendelünk. Mivel az Application objektum nem jelenik meg az Object Inspector listájában, ezért kézzel kell elkészítenünk az új metódust, és valamikor a program indításakor, például a form készítésekor, hozzá kell rendelnünk az `Application` objektum OnShowHint tulajdonságához.

Az új metódusnál definiálnunk kell néhány paramétert, mint a tipp szövegét tartalmazó karakterlánc, egy logikai jelző a tipp aktiválásáról, és egy struktúra, ami a többi információt tartalmazza:

```
TShowHintEvent = procedure (
  var HintStr: string;
  var CanShow: Boolean;
  var HintInfo: THintInfo) of object;
```

Mivel mindegyik paramétert hivatkozás szerint kapunk, ezért lehetőségünk van azok megváltoztatására. Az utolsó paraméter egy struktúra, ami tartalmazza a kontrolt, a tipp pozícióját, a színét, és még más információkat is:

```
THintInfo = record
  HintControl: TControl;
  HintPos: TPoint;
  HintMaxWidth: Integer;
  HintColor: TColor;
  CursorRect: TRect;
  CursorPos: TPoint;
end;
```

A struktúra értékeinek módosításával például megváltoztathatjuk a tippablak helyét még a megjelenítése előtt, ahogy azt a Phrases4 példában láthatjuk. Ha újra megnézzük a 12.6. ábrát, akkor láthatjuk, hogy a tipp nem a szokásos helyen, a kontrol alatt jelenik meg, hanem a kliens terület közepéhez közel. Ehhez a fenti TshowHintEvent típusban látott paraméterekkel egy új metódust kell a formhoz adnunk:

```
public
    procedure ShowHint (var HintStr: string;
      var CanShow: Boolean; var HintInfo: THintInfo);
```

Ezt a metódust a form készítésekor az Application objektum OnShowHint eseményéhez kell kapcsolnunk:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ...
  Application.OnShowHint := ShowHint;
end;
```

Ezután meg kell írnunk az új metódus kódját. A következő kód a fenti nagy címke, és a két listadoboz tippjeinek középre igazítását végzi:

```
procedure TForm1.ShowHint (var HintStr: string;
  var CanShow: Boolean; var HintInfo: THintInfo);
begin
  with HintInfo do
    if (HintControl = Label1) or
       (HintControl = ListBox1) or
       (HintControl = ListBox2) then
         HintPos := HintControl.ClientToScreen (Point (
           HintControl.Width div 2, HintControl.Height
                                                 div 2));
end;
```

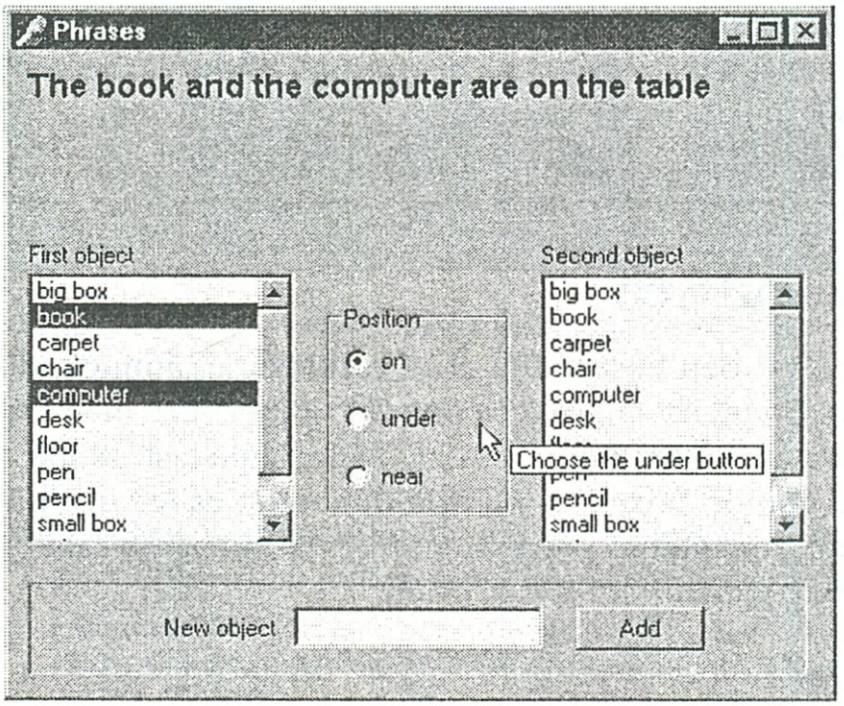
A kód egy kicsit nehezen érthető, mert meg kell határoznunk az általános kontrol, a HintInfo.HintControl, középpontját , és ezt a ClientToScreen metódussal le kell képeznünk magára a kontrolra. Ahogy azt már említettük a kód hatását a 12.6. ábrán láthatjuk.

Több tipp egy kontrolhoz

Tovább bővíthetjük a Phrases4 példát egy másik módon is. A form közepén lévő RadioGroup kontrolnak három rádiógombja van, amik nem önálló komponensek, hanem csak három rádiógomb klón, amiket a rádió csoportra rajzolunk. Mi van akkor, ha mindegyikhez külön tippet akarunk adni?

Erre a célra a THintInfo (tipp információ) rekord CursorRect (Kurzor Téglalap) mezőjét használhatjuk fel, ami a komponensnek azt a területét jelzi, amin belül

anélkül mozoghatunk a kurzorral, hogy a tipp eltűnne. Ha mindegyik rádió gombnak más területet és szöveget adunk meg, akkor gyakorlatilag három különböző tippünk lehet. Mivel a rádiógombok pontos helyének kiszámítása nem egyszerű feladat, ezért a rádió csoport területét egyszerűen három egyenlő részre bontottuk. Az egérmutató alatti rádiógomb szövegét ezután hozzáadjuk a tipp szövegéhez:



12.7. ábra *A Phrases4 példa rádió csoportja bemutatja a rádiógombonként külön tipp használatát*

```
procedure TForm1.ShowHint (var HintStr: string;
  var CanShow: Boolean; var HintInfo: THintInfo);
var
  RadioItem, RadioHeight: Integer;
  RadioRect: TRect;
begin
  with HintInfo do
    if (HintControl = Label1) ... // as before
    else
    if HintControl = RadioGroup1 then
    begin
      RadioHeight := RadioGroup1.Height div 3;
      RadioItem := CursorPos.Y div RadioHeight;
      HintStr := 'Choose the ' +
        RadioGroup1.Items [RadioItem] + ' button';
      RadioRect := RadioGroup1.ClientRect;
```

```
            RadioRect.Top := RadioRect.Top +
              RadioHeight * RadioItem;
            RadioRect.Bottom := RadioRect.Top + RadioHeight;
            // assign the hints rect and pos
            CursorRect := RadioRect;
            HintPos := RadioGroup1.ClientToScreen (
              RadioRect.BottomRight);
        end;
    end;
```

A kód utolsó része készíti el a tipp téglalapját, a komponens kliens területéből kiindulva, úgy, hogy a Top (felső) és Bottom (alsó) értékeket a megfelelő RadioGroup1 elem helyéhez módosítjuk. Ennek a téglalapnak a jobb alsó sarka határozza meg a tipp helyét, ahogy az a 12.7. ábrán is látható.

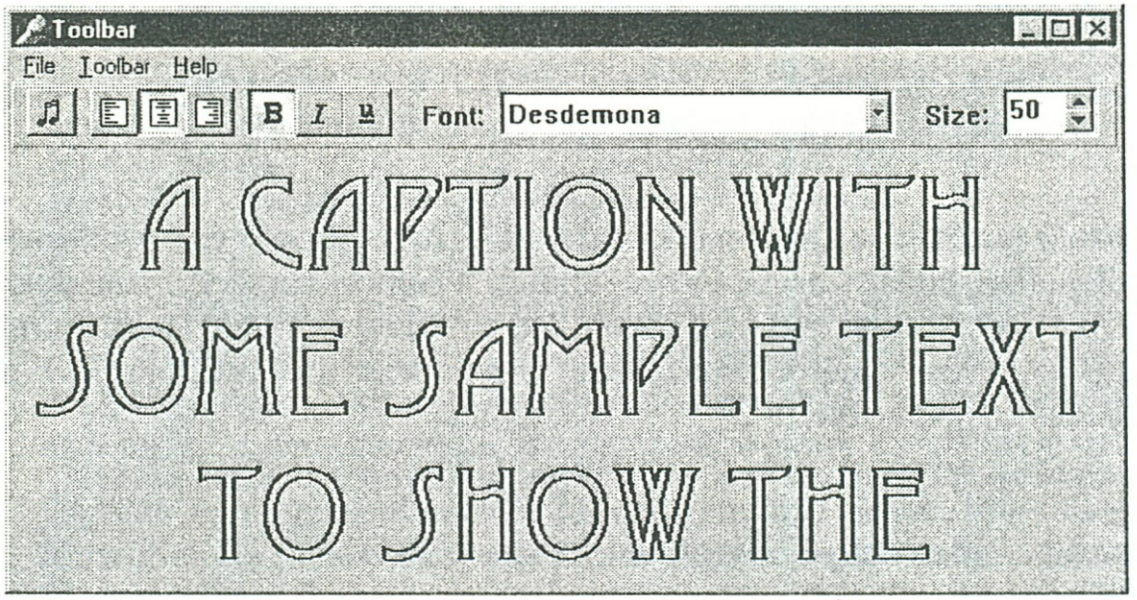
Szolgáltatások az eszköztáron

A Toolbar példában három lépésben készítettük el az eszköztárat, aminek különböző viselkedésű gombjai voltak, változó képei, és tippjei. Ezt még két irányba fejleszthetjük tovább, az egyik, hogy néhány új szolgáltatást adunk a panelhez, mint amilyenek a combo boxok, a legördülő menük, és így tovább, a másik pedig, hogy húzható, mozgatható eszköztárat készítünk, olyat, amit a form bármelyik oldalára átmozgathatunk futás közben.

Combo box eszköztáron

Az első bővítés a ComboBar példa, aminek az eszköztárában szerepel egy kombi panel is. Sok általános alkalmazás használ kombi paneleket stílusok, betűtípusok, betűméretek és más egyebek megjelenítésére. A programunk is ezt a vonalat követi, de egy kissé másként: kombi panelt használunk ugyan a betűtípus kiválasztására, de a betűméret kiválasztására egy UpDown (föl le) komponenssel összekötött szerkesztőmező szolgál. A programot futás közben a 12.8. ábrán láthatjuk.

Ennek a példának a formja hasonlít az előző verzióéra, a ToolBar3-éra, de néhány összetettebb funkció hiányzik. Nem szerepel a Size speed button, de van helyette egy kombi panel és egy szerkesztőmező, az UpDown komponenssel, és mindegyiknek saját címkéje van. A három új komponensnek és a címkéiknek saját tippjeik vannak. A formnak van még egy másik komponense is, egy előugró menü, amit a panelhez kapcsoltunk, és aminek majdnem ugyanazok a parancsai vannak, mint a menüsor Toolbar menüjének. Hogy ezt az új menüt minél kevesebb munkával működőképessé tegyük, a pontjai OnClick eseményeit az Object Inspector segítségével a főmenü azonos pontjainak metódusaira irányítottuk.



12.8.ábra *A ComboBar példa futás közben*

Az eredeti metódusok csak a Toolbar legördülő menüben állítják be a jelöléseket, az előugró menüben nem. Bár átírhatnánk, hogy kezeljék az előugró menüt is, mégis másik megoldást választottunk. Az előugró menü megjelenésekor lemásoljuk a főmenü megfelelő pontjainak jelöléseit:

```
procedure TToolbarForm.PopupMenu1Popup(Sender: TObject);
begin
  DisableSound2.Checked := DisableSound1.Checked;
  DisableStyles2.Checked := DisableStyles1.Checked;
end;
```

Az új kód többi része az eszköztár új komponenseire vonatkozik. A kombi panelt az alkalmazás kezdetekor töltjük fel úgy, hogy a Screen (képernyő) objektumból kimásoljuk az elérhető betűtípusok neveit, ahogy azt a tizenegyedik fejezetben szereplő FontGrid példában tettük:

```
procedure TToolbarForm.FormCreate(Sender: TObject);
begin
  ComboBox1.Items := Screen.Fonts;
  {select the current font}
  ComboBox1.ItemIndex :=
    ComboBox1.Items.IndexOf (Label1.Font.Name);
end;
```

Vegyük észre, hogy a metódus utolsó sorával a címke aktuális betűtípusát választottuk ki a kombi panelben. Ha a kombi panelben másik elemet választunk ki, akkor ennek pont az ellenkezője történik: A kombi panel aktuális elemének szövegét másoljuk a címke betűtípusához:

```
procedure TToolbarForm.ComboBox1Change(Sender: TObject);
begin
  Label1.Font.Name :=
    ComboBox1.Items [ComboBox1.ItemIndex];
end;
```

Valami hasonló történik az UpDown kontrolnál is. Az UpDown és a szerkesztőmező pillanatnyi értékét az UpDown kontrol Position (pozíció) tulajdonságával érhetjük el, és ezt másoljuk át a címke Font (betűtípus) tulajdonságának Size (méret) mezőjébe. Ez akkor történik, amikor a szerkesztőmező megváltozik:

```
procedure TToolbarForm.Edit1Change(Sender: TObject);
begin
  Label1.Font.Size := UpDown1.Position;
end;
```

Mozgatható eszköztár

A Toolbar példa másik továbbfejlesztése, hogy a felhasználó számára lehetővé tesszük, hogy az eszköztárat elmozgassa a képernyő más részeire. Néhány Windows alkalmazásban az eszköztárakat csak a menü alá, vagy az ablak aljára tehetünk, máshol lehetnek függőleges eszköztárak is, megint másoknál pedig az eszköztárból úgynevezett toolbox-ot, kis lebegő ablakot készíthetünk. A következő példa, a DragTool, ezek mindegyikét bemutatja.

Ehhez az eszköztárként működő panelnek támogatnia kell az automatikus húzást, a címkének pedig fogadnia kell tudni a húzást. Először a Panel1 DragMode (húzásmód) tulajdonságát állítsuk dmAutomatic-ra, ezután a formot betöltő címkének készítsünk egy OnDragOver eseménykezelőt, ami az Accept (elfogad) paramétert igazra állítja. Az eszköztár lehetséges pozícióinak jelölésére a TAlign (igazítás) adattípust használjuk, amit a VCL a következőképp deifniál:

```
type
  TAlign = (alNone, alTop, alBottom,
    alLeft, alRight, alClient);
```

Az alNone érték fogja jelezni, hogy az eszköztárunk lebegő ablak formájú. Ez az adattípus a GetBarPos függvény visszatérési értéke, ami a koordinátákat jelölő X és Y paraméterektől függő kódot ad vissza:

```
function TToolbarForm.GetBarPos (X, Y: Integer):
TAlign;
begin
  if X < DragSize then
    GetBarPos := alLeft
  else if Y < DragSize then
    GetBarPos := alTop
```

```
    else if X > ClientWidth - DragSize then
      GetBarPos := alRight
    else if Y > ClientHeight - DragSize then
      GetBarPos := alBottom
    else
      GetBarPos := alNone;
  end;
```

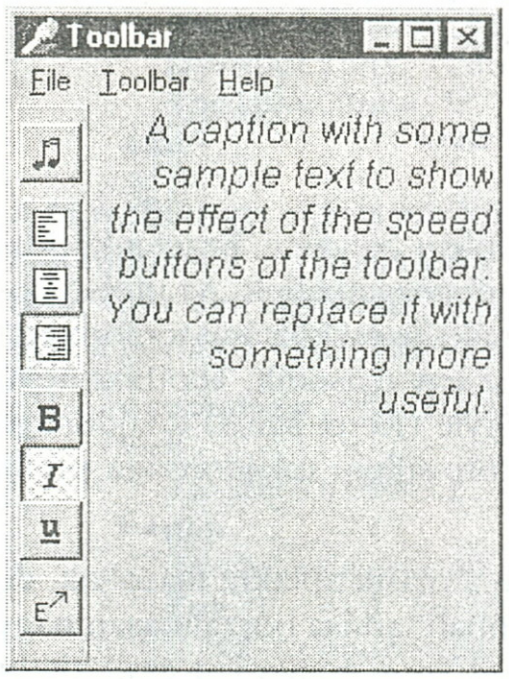
Ebben a kódban a DragSize (húzási méret) a program által definiált konstans, ami lehetőleg kisebb az eszköztár szélességénél – vagy magasságánál. Ha a húzás befejeződött, akkor az eszköztárat a megfelelő pozícióba kell vinnünk, és ez a program érdekes része, de egyszerűbb, mint azt elsőre gondolnánk.

Kezdjük egy egyszerű esettel. Mi kell tennünk ahhoz, hogy az eszköztárat a form tetejéről az aljára vigyük? Ehhez elég, ha a panel Align (igazítás) tulajdonságát átállítjuk:

```
Panel1.Align := alBottom;
```

Arra azonban figyeljünk, hogy a form többi komponensét csak azért hagyhatjuk figyelmen kívül, mert csak egyetlen, a kliens területhez igazított címkénk van, az pedig a form teljes területét eltakarja az eszköztár által elfoglaltakon kívül. Hogy a Panel1 visszakerüljön fentre ennek az ellenkezője szükséges. Ha az eszköztárat máshova visszük, ahogy az a 12.9. ábrán is látható, akkor ugyanezt kell tennünk, a GetBarPos függvény visszatérési értékét felhasználva:

```
ReqPos := GetBarPos (X + Label1.Left, Y + Label1.Top);
Panel1.Align := ReqPos;
```



12.9. ábra *A DragTool példában a felhasználó az eszköztárat elmozgathatja a form szélére*

Hogy a panelt függőlegessé alakítsuk, a probléma nem a panelnél jelentkezik, hanem a SpeedButtonoknál, ezek mindegyikét új helyre kell mozgatnunk, máskülönben csak az első látszódna. Ez megint összetettebbnek tűnik, mint amilyen. Hogy egy gombot a vízszintes sorból a függőlegesbe tegyünk elég, ha egyszerűen felcseréljük a Top (felső) és a Left (bal) koordinátáit. Rövid gondolkodás után beláthatjuk, hogy miért van ez így. Ahhoz, hogy az így átrendezett gombokat visszarendezzük, ahhoz ugyanezt kell újra megtennünk. Az ehhez szükséges kódot a RotateSpeedbar eljárás tartalmazza:

```
procedure TToolbarForm.RotateSpeedbar;
var
  I, X, Y: Integer;
begin
  for I := 0 to Panel1.ControlCount - 1 do
  begin
    {reverse X and Y}
    X := Panel1.Controls.Top;
    Y := Panel1.Controls.Left;
    Panel1.Controls.Top := Y;
    Panel1.Controls.Left := X;
  end;
end;
```

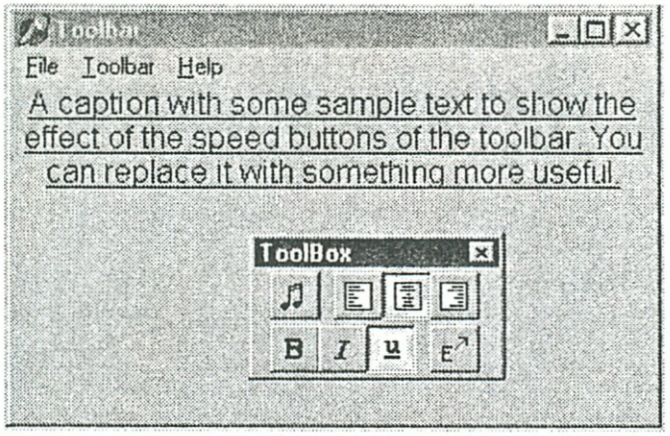
A metódus a Controls (kontrolok) tömbön megy végig a ControlCount (kontrolszám) alapján, és az eszköztár összes gombjának felcseréli az X és Y koordinátáit. Ez természetesen csak négyzetes kontrolokkal működik jól; ha kombi panel is van az eszköztáron, akkor a végeredmény nem lesz helyes. Mikor kell meghívnunk ezt a metódus? Csak akkor, ha a régi és az új eszköztár közül az egyik vízszintes, a másik pedig függőleges:

```
if (ReqPos in [alTop, alBottom]) <>
  (BarPos in [alTop, alBottom]) then
  RotateSpeedBar;
```

Ebben a vizsgálatban a BarPos az eszköztár jelenlegi helyzete. Ezzel a kóddal az eszköztárunkat a form mind a négy oldalára elmozgathatjuk. Az utolsó dolog, amit még meg kell tennünk, hogy a felhasználó az eszköztárat toolbox-szá alakítsa, amint az a 12.10. ábrán is látszik. Ehhez egy második, ToolBox nevű formot adtunk a projekthez, aminek a BorderStyle (Keret stílus) tulajdonága bsToolWindow (eszköz ablak), a FormStyle (formstílus) tulajdonsága pedig fsStayOnTop (mindig felül).

Először arra gondoltunk, hogy a panel gombjait a másik ablakra másoljuk, de azután rájöttünk, hogy a tulajdonságok szinkronban tartása borzalmasan nagy feladat lenne, ezért ezt a formot üresen hagytuk, és a gombokat csak akkor másoljuk át, amikor szükség van rájuk. Ha a húzás a form középső részén fejeződik be, akkor megjelenítjük a toolboxot, elrejtjük az eszköztárat,

elforgatjuk a gombokat, ha függőlegesen álltak, és meghívjuk a MoveButtons (gombok mozgatása) metódust.



12.10. ábra *A DragTool példa eszköztára*

Ez a kód még igazra állítja a BoxOn privát logikai mezőt, ami azt jelzi, hogy a toolbox látható:

```
if ReqPos = alNone then
begin
  Panel1.Visible := False;
  BoxOn := True;
  if BarPos in [alLeft, alRight] then
    RotateSpeedbar;
  {move buttons to the toolbox}
  MoveButtons (ToolBox);
  ToolBox.Show;
end
```

A MoveButtons eljárás két különböző feladatot hajt végre: először átmozgatja a gombokat a paraméterként átadott TWinControl-ra, ezután a két sorba rendezi, vagy újra egy sorba hozza azokat – ezért használhatjuk ugyanezt a metódust a gombok visszaállítására is:

```
procedure TToolbarForm.MoveButtons (Win: TWinControl);
var
  I, J: Integer;
begin
  {move the speed buttons to a new parent component}
  for I := 0 to ComponentCount - 1 do
    if Components is TSpeedButton then
      TSpeedButton (Components).Parent := Win;
    if Win = ToolBox then
    begin
      {place the buttons on two lines}
```

```
          for J := 0 to Win.ControlCount - 1 do
            if Win.Controls [J].Left > 110 then
            begin
              Win.Controls [J].Left :=
                Win.Controls [J].Left - 112;
              Win.Controls [J].Top := 30;
            end;
        end
        else
        begin
          {place back the buttons on one line}
          for J := 0 to Win.ControlCount - 1 do
            if Win.Controls [J].Top = 30 then
            begin
              Win.Controls [J].Left :=
                Win.Controls [J].Left + 112;
              Win.Controls [J].Top := 2;
            end;
        end;
    end;
```

Ha a toolbox megjelent, többet nem húzhatjuk vissza a form szélére eszköztárnak. Viszont ha becsukjuk az ablakát, akkor az eszköztár újra megjelenik a régi helyén. Ezt a toolbox OnClose (bezáráskor) eseménykezelője végzi:

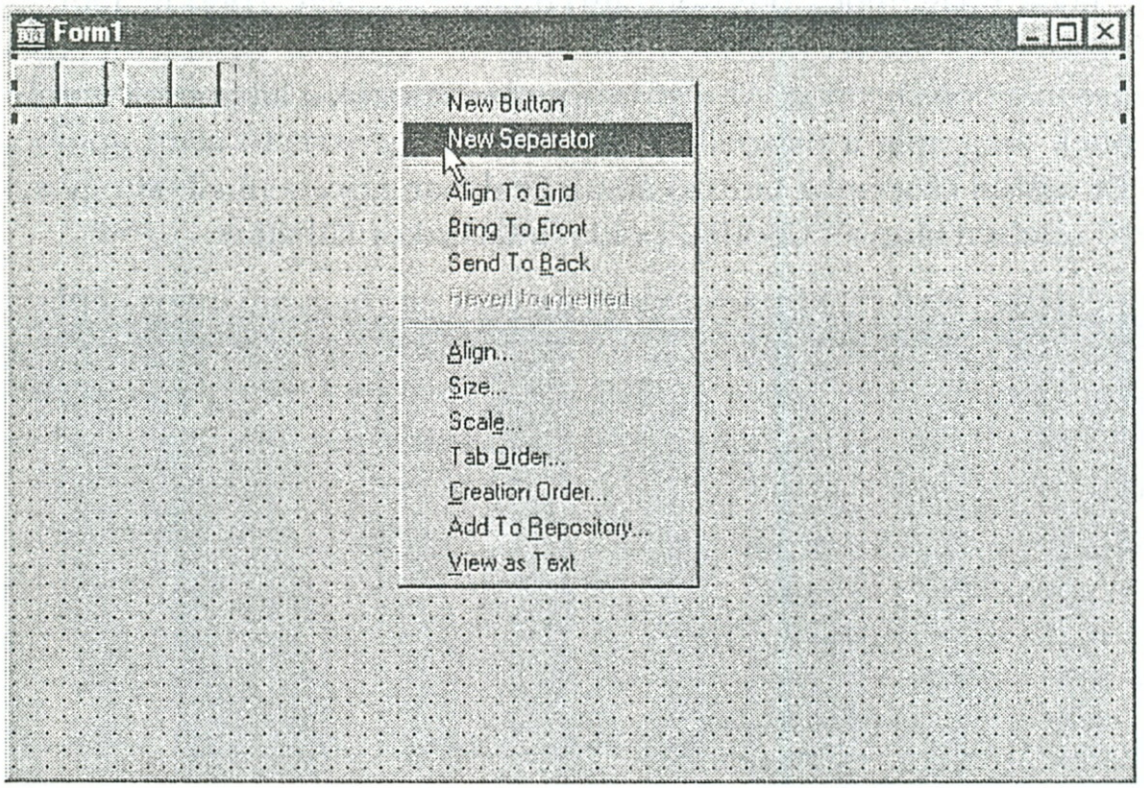
```
procedure TToolBox.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  ToolbarForm.Panel1.Visible := True;
  ToolbarForm.BoxOn := False;
  ToolbarForm.MoveButtons (ToolbarForm.Panel1);
  if ToolbarForm.BarPos in [alLeft, alRight] then
    ToolbarForm.RotateSpeedbar;
end;
```

A Windows 95-ös Toolbar (Eszköztár) kontrol

A Delphi 3-ban jelent meg az Eszköztár komponens, ami az ennek megfelelő Windows 95 szabványos kontrolt takarja. Ez egy eszköztár saját gombokkal, és a házi eszköztárunkhoz képest néhány extra szolgáltatással. Erre a komponensre jó példa a Microsoft Internet Explorer legújabb változata, ahol Toolbarokat találhatunk Coolbarokon, ez utóbbi egy másik új komponens, amiről hamarosan szó lesz.

Az új komponens megismerését kezdjük azzal, hogy elkészítjük a már ismerős Toolbar példát, úgy, hogy panel helyett ezt a komponenst használjuk. Ez nagyon egyszerű lesz, induljunk el egy teljesen üres formmal, és tegyünk rá egy Toolbar komponenst a komponenspaletta Windows 95 oldaláról, és máris van

egy üres eszköztárunk. Gombokat úgy adhatunk hozzá, hogy a komponens helyi menüjéből kiválasztjuk az idevágó két menüpont egyikét, a **New Button**-t (új gomb) vagy **New Separator**-t (új elválasztó). Eszköztárra láthatunk példát a 12.11. ábrán.



12.11. ábra *Eszköztár készítéséhez tegyük a megfelelő komponenst a formra, és használjuk a helyi menüjének két különleges parancsát*

A Toolbaron TToolButton (eszköztár gomb) osztályú objektumok vannak, ezek ugyanolyan belső objektumok, mint a MainMenu komponensnél a TMenuItem. Ezeknél az objektumoknál egy tulajdonság, a Style (stílus), határozza meg a viselkedésüket:

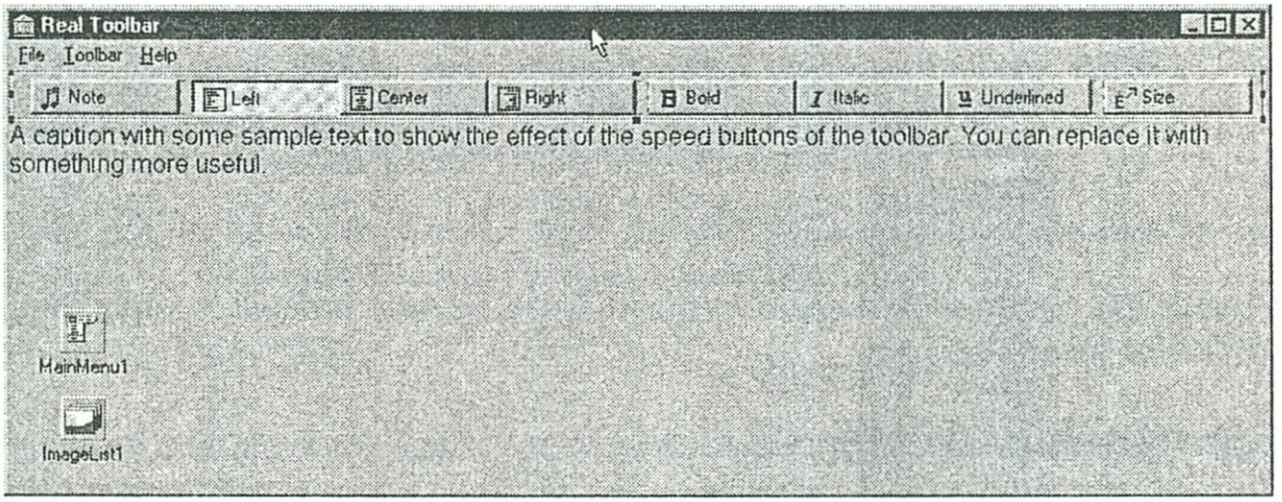
- A tbsButton stílus a hagyományos nyomógombot jelöli
- A tbsCheck stílus jelölőnégyzetet, vagy rádiógombot jelez. Rádiógomb akkor, ha a saját, elválasztók által meghatározott blokkjában több ilyen tulajdonságú gombbal van csoportosítva (Grouped tulajdonság).
- A tbsDropDown stílus legördülő gombot jelent, ami egyfajta jelölőnégyzet. A legördülő résznek azonban nincs közvetlen támogatása, ezt más komponens használatával kell előkészíteni.
- A tbsSeparator stílus elválasztót jelöl.

A következő lépés a képek hozzárendelése a gombokhoz. Ehhez szükség van a formon egy vagy több ImageList (képlista) komponensre. A képeket az Imagelist-be kell betöltenünk, majd az Imagelistet az Images tulajdonságon keresztül össze kell kötnünk az eszköztárral. Alapbeállítás szerint a képek a megjelenés sorrendjében kerülnek a gombokra, de ezt könnyen megváltoztathatjuk a gombok ImageIndex (képsorszám) tulajdonságával. Több ImageListet is készíthetünk, és ezeket hozzárendelhetjük az eszköztár DisabledImages és HotImages tulajdonságaihoz. Az első csoportba tartozók a kikapcsolt gombóknál jelennek meg, míg a másodikba tartozók az egérmutató alatti gombra kerülnek. Ez utóbbi csoportba tartozó ábrák általában színesebbek, ahogy azt a következő részben tárgyalt CoolBar példa futtatásakor láthatjuk.

A RealBar példában lévő eszköztáron azonban csak egy hagyományos képlista van. Ez a példa, a már látott példa új változata, amiben valódi eszköztár – egy TToolbar komponens- van. Ennek a példának a tervezés közbeni formját láthatjuk a 12.12. ábrán, itt pedig a szöveges leírásának néhány részlete következik:

```
object ToolBar1: TToolBar
  AllowAllUp = False
  AutoSize = True
  Flat = True
  Images = ImageList1
  ShowCaptions = True
  ShowHint = True
  object TbNote: TToolButton
    Hint = 'Play Note'
    Caption = 'Note'
    ImageIndex = 0
    OnClick = TbNoteClick
  end
  object TbSeparator1: TToolButton
    Style = tbsSeparator
  end
  object TbLeft: TToolButton
    Hint = 'Align Left'
    Caption = 'Left'
    Down = True
    Grouped = True
    ImageIndex = 1
    Style = tbsCheck
    OnClick = TbLeftClick
  end
  object TbCenter: TToolButton...
  object TbRight: TToolButton...
  object TbSeparator2: TToolButton...
  object TbBold: TToolButton...
```

```
    object TbItalic: TToolButton...
    object TbUnderlined: TToolButton...
    object TbSeparator3: TToolButton...
    object TbSize: TToolButton...
      Hint = 'Resize Text'
      Caption = 'Size'
      ImageIndex = 7
      Wrap = True
      Style = tbsCheck
      OnClick = TbSizeClick
    end
  end
```

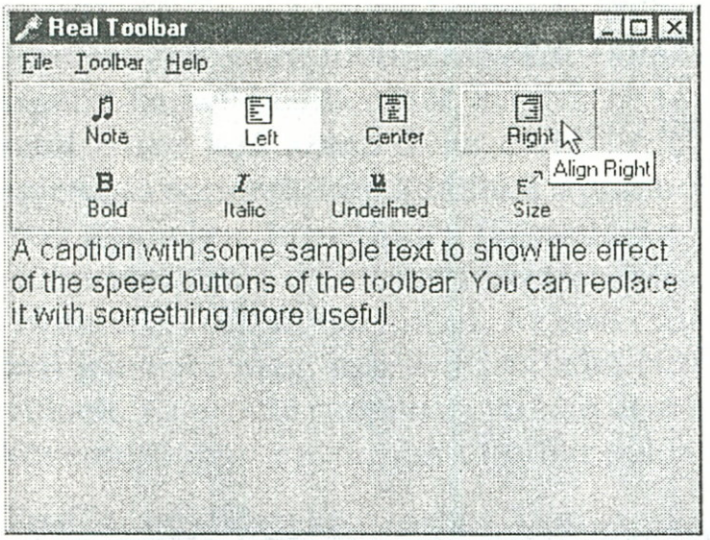


12.12. ábra *A RealBar példa Toolbar komponense tervezés közben*

A Toolbar komponens nagyon rugalmas. A gombokra tehetünk feliratokat, de nem kötelező, ha mégis teszünk, akkor ezek kerülhetnek a kép baloldalára, vagy alulra, és választhatunk a hagyományos, és az új, lapos gombok között. A RealBar példa Toolbar legördülő menüje lehetővé teszi, hogy ezeket a tulajdonságokat futás közben is változtathassuk, amint az a 12.13. ábrán látható.

Ennek a három gombnak a kódja csak átállítja az eszköztár komponens néhány tulajdonságát, azután a megfelelő menüpontoknál beállítja a szükséges jelöléseket. Az első parancs ki is kapcsolja a másodikat, hiszen a nem látható címkék mozgatásának nincs értelme:

```
procedure TRealBarForm.Captions1Click(Sender: TObject);
begin
  Toolbar1.ShowCaptions := not Toolbar1.ShowCaptions;
  Captions1.Checked := Toolbar1.ShowCaptions;
  CaptionBelow1.Enabled := Toolbar1.ShowCaptions;
end;
```



12.13. ábra *A RealBar példa Toolbar komponense többféle módon megváltoztatható, ami által nagyon különbözhet az eredeti tervezés közbeni változattól, ami a 12.12. ábrán látható*

A másik két parancsnak nagyon egyszerű kezelője van, aminek a kódja megtalálható a kísérőlemezen.

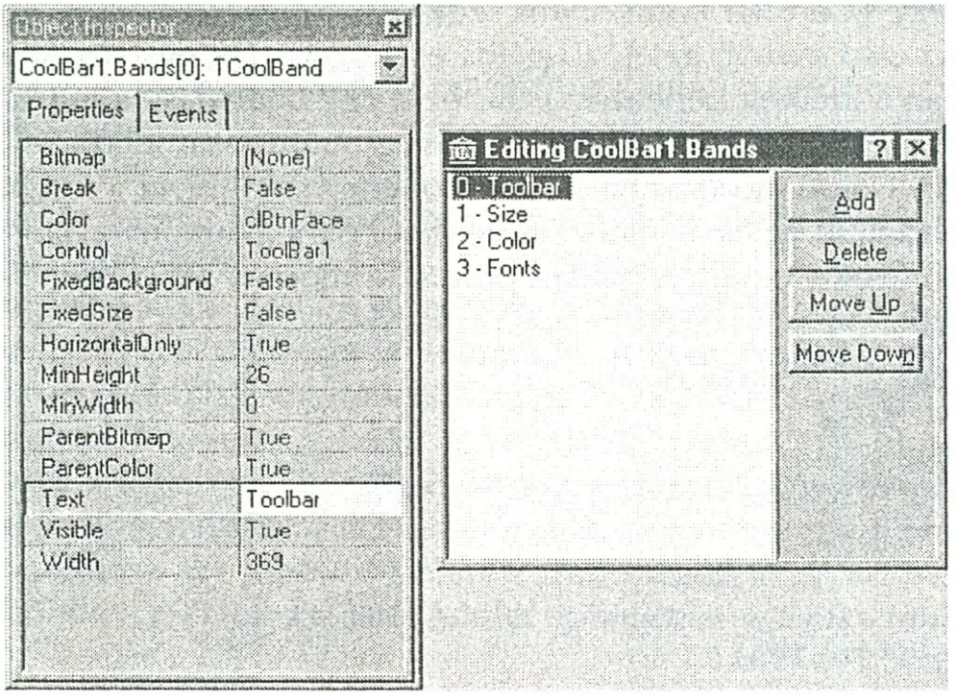
Egy remek eszköztár

A Toolbar komponens mellet a Delphi 3-ban szerepel még a CoolBar komponens is, ami a Microsoft Internet Explorer 3.0-ban megjelent, és a felújított COMCTRLS.DLL részét képező vadonatúj Windows kontrol.

A CoolBar komponens gyakorlatilag TCoolBand objektumok gyűjteménye. Az eszköztár gomboktól eltérően ezek az objektumok nem jelennek meg önálló objektumokként a formon, hanem csak alpontok. Az Object Inspectorban csak akkor jelennek meg, ha a CoolBar Bands (sávok) tulajdonság szerkesztőjét választjuk ki, ahogy az a 12.14. ábrán látszik. Könnyedén készíthetünk egy, vagy több sávot, és beállíthatjuk a tulajdonságaikat.

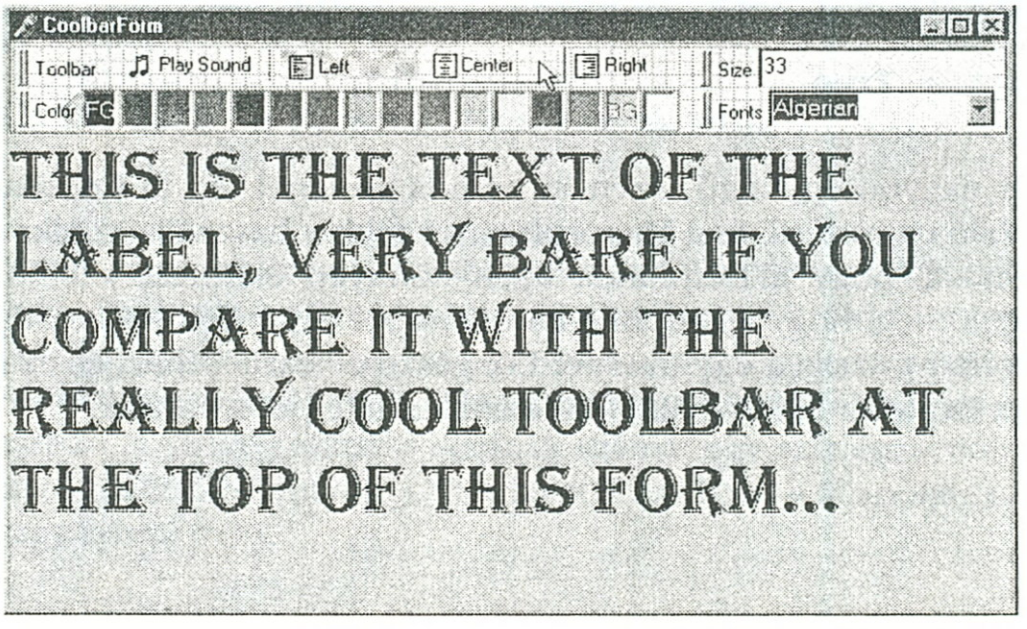
A CoolBar komponens használata egyszerű: háttérként beállíthatunk egy képet, hozzáadunk néhány sávot a Bands tulajdonsággal, ezután mindegyk sávhoz hozzárendelhetünk már létező komponenseket, vagy komponens tárolókat. Bármilyen ablak alapú kontrolt – nem grafikus kontrolt – használhatunk, de csak néhány jelenik meg helyesen. Ha képet szeretnénk a CoolBar hátterében, akkor részben átlátszó kontrolokra van szükségünk.

A CoolBar-okon legáltalánosabban használt komponens a Toolbar – amit teljesen átlátszóvá is tehetünk – de kombi panelek, szerkesztőmezők és animációs kontrolok is gyakran előfordulnak. A felépítésüket gyakran az Internet Explorer felülete sugallja, az első olyan Microsoft alkalmazásé, ami a CoolBar komponenshez hasonló eszköztárat használ.



12.14. ábra *A CoolBar komponens Bands tulajdonságának szerkesztője az Object Inspectorral együtt működik*

Tehetünk minden sorba egy sávot, vagy az összest egy sorba, úgy, hogy mindegyik a használható rész kis részén helyezkedik el, és automatikusan megnő, ha a címére kattintunk. Ezt az új komponenst sokkal könnyebb használni, mint elmagyarázni. Próbálkozzunk vele önállóan, vagy kövessük a lenti leírást, amiben a Toolbar példa új változatát készítjük el a CoolBar kontrol segítségével. Ennek a formját láthatjuk futás közben a 12.15. ábrán.



12.15 ábra *A CoolBar példa formja futás közben*

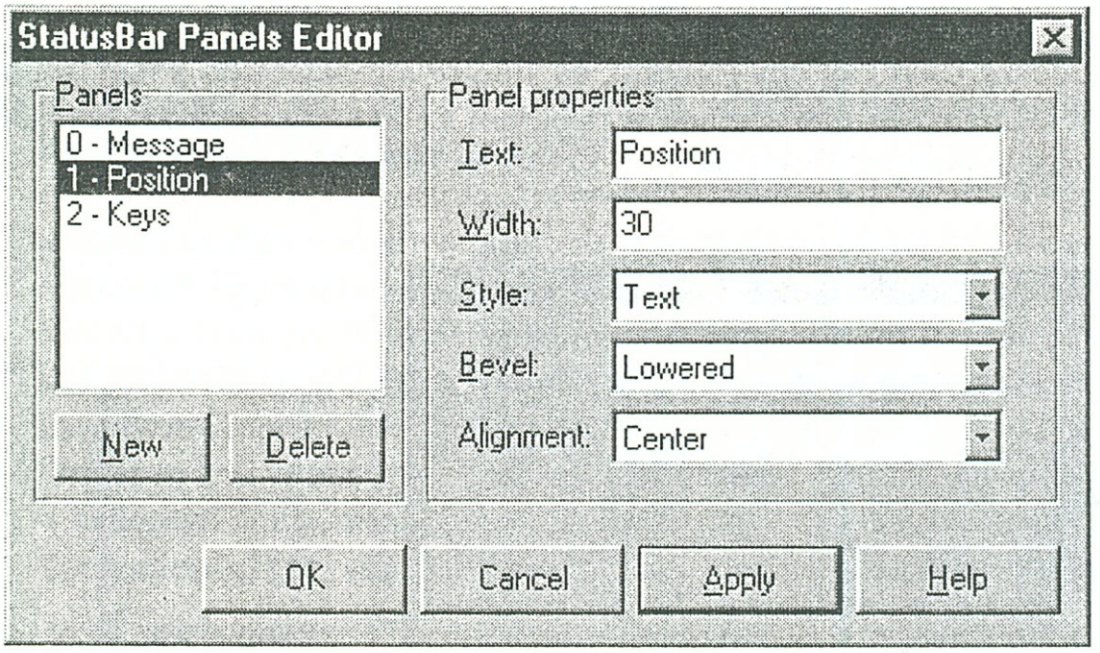
A CoolBar példában egy TCoolBar komponens szerepel, négy sávval, minden sorban kettő–kettő. Az első sávban az előző példa eszköztárának egy részlete szerepel, most azonban a kiemelt képekhez is tartozik egy ImageList; a másodikban egy szerkesztőmező található, amivel a szöveg betűtípusát állíthatjuk be; a harmadikban egy ColorGrid (színháló) komponens van, amivel a betűk és a háttér színét választatjuk ki; és az utolsó egy kombi panelt tartalmaz, amiben az elérhető betűtípusok vannak. A példa CoolBar komponensének szöveges leírása a következő:

```
object CoolBar1: TCoolBar
  AutoSize = True
  BandBorderStyle = bsNone
  Bands = <
    item
      Control = ToolBar1
      HorizontalOnly = True
      Text = 'Toolbar'
    end
    item
      Control = Edit1
      Text = 'Size'
    end
    item
      Control = ColorGrid1
      HorizontalOnly = True
      Text = 'Color'
    end
    item
      Control = ComboBox1
      Text = 'Fonts'
    end>
  BorderWidth = 1
  Bitmap.Data = {...}
  ShowHint = True
end
```

A program kódja nagyon egyszerű, ezért itt nem is szerepel. Az új CoolBar komponens valóban nagyszerű, és a Microsoft egyre többet használja az alkalmazásaiban, ezért az, hogy a Delphiben ilyen könnyen elérhető, jelentős előny. Itt jegyezzük meg, hogy a Delphi 3-ban szerepel egy példa – CoolStuf néven –, ami CoolBar-t használ egy Internet Böngésző elkészítéséhez, és ez a Microsoft Internet Explorer felhasználói felületének szinte pontos mása.

Státuszsor készítése

Státuszsort készíteni még egyszerűbb, mint eszköztárat. A Delphiben szerepel egy StatusBar (státuszsor) komponens, ami a megfelelő Windows 95 kontrolra épül. Bár használhatnánk alulra igazított Panel komponenst is erre a célra, mégis javasolt a StatusBar komponenst használata. Ez a komponens szinte teljesen panelként funkcionál, ha a SimplePanel (egyszerű panel) tulajdonságát igazra állítjuk. Ebben az esetben a SimpleText (egyszerű szöveg) tulajdonság segítségével szöveget irathatunk ki. A komponens valódi előnye azonban az, hogy több alpanelt definiálhatunk rá a Panels (panelek) tulajdonság szerkesztőjével, ahogy az a 12.16. ábrán látható. Mindegyik alpanelnek megvannak a saját grafikus tulajdonságai, amiket a szerkesztő segítségével állíthatunk. Másik előnye a státuszsor komponensnek, hogy van egy úgynevezett „méret-fogó" területe a sáv jobb alsó részén, ami megkönnyíti a form átméretezését. Ez a Windows 95 felhasználói felület tipikus eleme.



12.16. ábra *A StatusBar Panels szerkesztője*

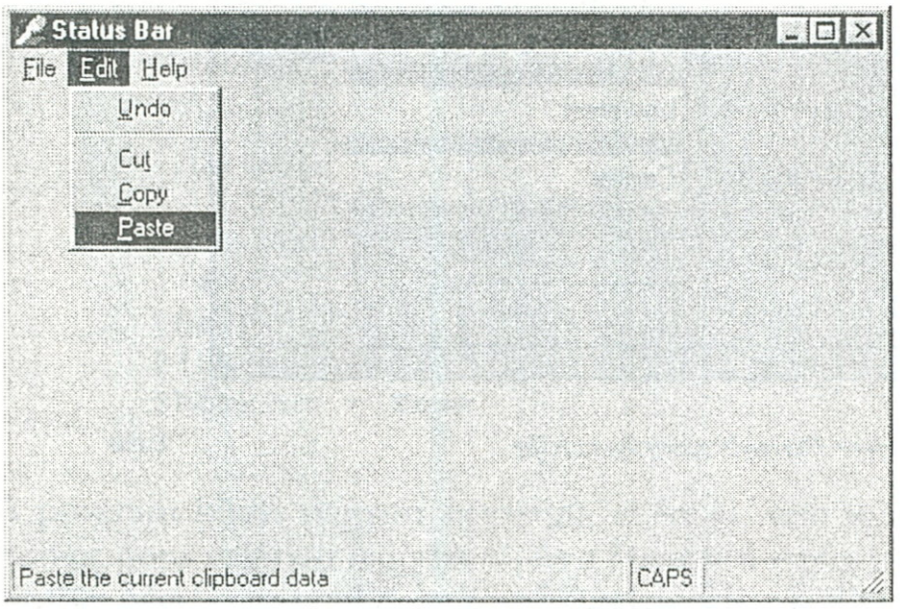
A státuszsornak számos felhasználási lehetősége van. A leggyakrabban előforduló, hogy az éppen kiválasztott menüpontról rövid ismertető jelenik meg. Emellett gyakran látható rajta információ a program státuszáról – innen a neve. Például megjeleníthetjük rajta az egérmutató pozícióját egy grafikus alkalmazásban, a szöveg aktuális sorának számát egy szövegszerkesztőben, a különböző rögzítő billentyűk – NUM LOCK, CAPS LOCK – állapotát, a dátumot, az időt, vagy bármi mást.

Menü tippek a státuszsoron

Az első példánkban, a Status1-ben, a státuszsor a menüpontok leírását, és a CAPS LOCK gomb állapotát fogja megjeleníteni. Íme a StatusBar komponens tulajdonságai:

```
object StatusBar1: TStatusBar
    Panels = <
      item
        Width = 300
      end
      item
        Width = 36
      end
      item
        Width = 100
      end>
    SimplePanel = False
  end
```

A formon természetesen van egy főmenü, és amikor a felhasználó a menüben mozog, hogy kiválasszon egy elemet, a program az aktuális parancs leírását megjeleníti a státuszsoron, ahogy az a 12.17. ábrán látható.



12.17. ábra *A Status1 példa egyszerű státuszsora az aktuális menüpont leírását mutatja*

Ehhez két dolgot kell tennünk. Először a főmenü minden menüjéhez és menüpontjához írjunk egy rövid leírást a Hint tulajdonságukba. Másodszor írjunk egy rutint, ami az alkalmazás OnHint eseményét kezeli. Ehhez kézzel kell

készítenünk egy új metódust a formhoz, majd a program indulásakor – például form keletkezésekor – hozzá kell rendelnünk az `Application` objektum OnHint eseményéhez:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnHint := ShowHint;
end;
```

A form interfészéhez adjuk hozzá a következő definíciót:

```
procedure ShowHint(Sender: TObject);
```

Ez az eljárás a státuszsorba másolja az alkalmazás Hint tulajdonságát, ami átmenetileg a kiválasztott elem tippjét tartalmazza:

```
procedure TForm1.ShowHint(Sender: TObject);
begin
  StatusBar1.Panels[0].Text := Application.Hint;
end;
```

A menük hatását jelző tippek megjelenítése a státuszsoron ennyiből áll. Ahhoz, hogy a CAPS LOCK, vagy bármely másik billentyű státuszát megjelenítsük, meg kell hívnunk a GetKeyState (billentyű állapot lekérdezés) API függvényt, ami egy állapotszámot ad vissza. Ha ennek a számnak a legalsó bitje egyes – vagyis a szám páratlan – akkor a gomb le van nyomva. Mikor kell megnéznünk ezt az állapotot? Egyrészt megtehetnénk akkor, amikor a felhasználó lenyom egy billentyűt a formon, vagy használhatnánk egy időzítőt, és ellenőrizhetnénk másodpercenként. Ez a második megközelítés a jobb, mert a felhasználó megnyomhatja a CAPS LOCK-ot egy másik alkalmazásban is, és nekünk ezt is helyesen kell jeleznünk. Az időzítő használata azonban megnöveli a válaszidőt a billentyű lenyomásokra, ezért írtunk egy egyszerű CheckCapsLock (Caps Lock ellenőrzése) eljárást, amit mindkét eseménykezelőből meghívunk:

```
procedure TForm1.CheckCapslock;
begin
  if Odd (GetKeyState (VK_CAPITAL)) then
    StatusBar1.Panels[1].Text := 'CAPS'
  else
    StatusBar1.Panels[1].Text := '';
end;
procedure TForm1.FormKeyDown(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  CheckCapslock;
end;
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  CheckCapslock;
end;
```

Speed Button tippek a státuszsoron

Ezt az alkalmazást könnyedén kibővíthetjük egy eszköztárral, és megírhatjuk, hogy a gombok leírásai megjelenjenek, amikor az egérrel rájuk állunk. Emlékezzünk arra, hogy a gombok Hint tulajdonságait már felhasználtuk a sárga lebegő tippek megjelenítésére. Akkor hogyan használhatjuk ugyanazt a tulajdonságot két különböző dologra? Furcsa lehet, de ennek van közvetlen megoldása: írjunk egy egyszerű karaktersorozatot, amit egy elválasztóval, a | karakterrel, két részre vágunk. Például használhatjuk a tipp tulajdonság értékénél a következőt:

```
'Súgó|A súgó rendszer elõhívása'
```

A karakterlánc első része a lebegő tippekhez kerül, a második fele pedig a státuszsorra. Ez automatikusan történik, ha a státuszsor első verziójánál leírt kódot használjuk.

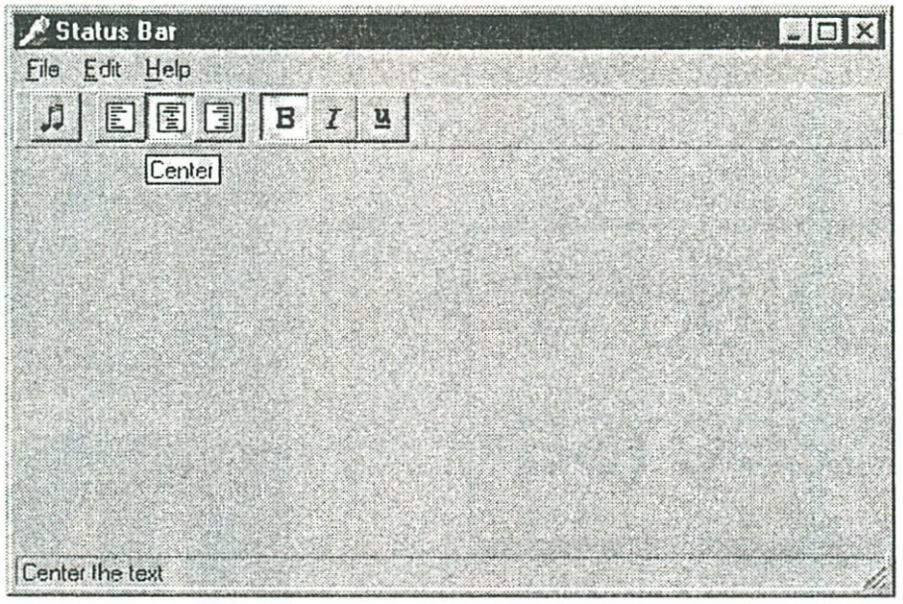
Megjegyzés

Ha a kontrol tippje két részből áll, akkor a GetShortHint, és a GetLongHint metódusokkal juthatunk hozzá az első – rövid –, illetve a második – hosszú – részéhez annak a paraméterként megkapott karakterláncnak, ami általában a Hint tulajdonság értéke.

Például a Státusz2 példában lemásoltuk a Status1 formját és kódját; ezután egy régebbi alkalmazásból átvettük az eszköztárat, eltávolítottuk a státuszsor alpaneljait, és a SimplePanel tulajdonságát igazra állítottuk, ezután a ShowHint metódus kódját a következőre változtattuk:

```
procedure TForm1.ShowHint(Sender: TObject);
begin
  StatusBar1.SimpleText := Application.Hint;
end;
```

Végül beállítottuk a grafikus gombok Hint tulajdonságait, és a kód megváltoztatása nélkül egyszerre látható mindkét tipp, ahogy az a 12.18. ábra is mutatja.



12.18. ábra *A Status2 példa képe, ahol a speed buttonoknak tippjük és státusz leírásuk is van*

Tipp

A fenti példa elkészítéséhez egy másik alkalmazásról másoltunk át egy komponens csoportot. Ez, ahogy a második fejezetben is láthattuk, nagyon egyszerű. Nyissuk meg a forrás alkalmazást, válasszuk ki a komponenseket – esetünkben a panelt, és néhány rajta lévő gombot – másoljuk a Vágólapra, menjünk a célformra, és illesszük be. Másik megoldásként használhattunk volna komponens sablonokat is.

Hogyan tovább?

Ebben a fejezetben átnéztük az egyszerű SDI alkalmazások eszköztárának és státuszsorának definícióját. Megnéztük, hogyan használhatjuk a megfelelő Windows 95 kontrolokat – a ToolBar, CoolBar és StatusBar Delphi komponensek formájában – vagy hogyan készíthetjük el ezeket egy egyszerű Panel és néhány SpeedButton segítségével.

A Delphiben három különböző módszer is kínálkozik eszközár vagy státuszsor készítésére az Application Expert (alkalmazás varázsló), vagy valamelyik alkalmazás sablon használatával. Amikor ezek a projektek elkészülnek, már működő eszköztáruk és státuszsoruk van.

Gondolhatjuk úgy, hogy ez a fejezet az első lépés a profi alkalmazások felé. A további lépések a következő fejezetekben vannak, de már most is eleget

13. FEJEZET

Több form és párbeszédablak

E fejezet tartalma:

- Párbeszédablakok vagy formok?
- Második form hozzáadása a programunkhoz
- Modális és nem modális formok
- Párbeszédablakok készítése
- Előre definiált párbeszédablakok használata
- Bővíthető párbeszédablakok
- Névjegy készítése
- Polimorf formok

Több form és párbeszédablak

A könyvben eddig látott programok egy formból álltak, és, ugyan lehetséges teljesen működő alkalmazást készíteni egyetlen formból és néhány más komponensből, de a legtöbb program több formból áll. Az általános alkalmazásoknak van egy főablakuk, lebegő eszköztáraik, palettáik, és több, különböző módon meghívható párbeszédablakuk. A bonyolultabb alkalmazások az MDI felépítést követik, vagyis egy keret ablak kliens területén több gyermekablakot használnak. Az MDI alkalmazásokról a tizenötödik fejezetben lesz szó, ez a fejezet a több mint egy formból vagy párbeszédablakból álló alkalmazásokkal foglalkozik.

A párbeszédablakok nem számítanak újdonságnak, hiszen már eddig is ezeket használtuk a színek, betűtípusok és állományok kiválasztására, valamint volt néhány üzenet ablakunk is, amit a MessageDlg függvénnyel készítettünk. Ebben a fejezetben megnézzük, hogyan készíthetünk saját párbeszédablakokat.

Párbeszédablakok vagy formok?

Mielőtt belekezdenénk a több formos vagy a felhasználó által definiált párbeszédablakos alkalmazások ismertetésébe, először nézzük meg a két lehetőség általános leírását, és a különbségeiket.

Már eddig is láthattuk a kapcsolatot a felhasználó szemszögéből Delphi formjai, és az ablakok között – erről a 10. fejezetben volt szó. Mivel a párbeszédablakok is egyfajta ablakok, ezért arra gondolhatunk, hogy ezek is formokra épülnek. Ez valóban így van; miután egy második formot adtunk a programunkhoz, akkor azt megjeleníthetjük akár formként, akár párbeszédablakként. A színfalak mögött, a Delphiben és a Windowsban kissé más játszódik le, de ez számunkra nem fontos.

A felhasználóknak a párbeszédablakokról a modális ablakok jutnak eszükbe – olyan ablakok, amiket miután megkapják a fókuszt, be kell zárni, mielőtt visszamehetnénk a főablakra. Ez igaz az üzenetablakokra, és általában a párbeszédablakokra is, ám ezek lehetnek nem modálisak is. Ha ebből arra következtetünk, hogy a párbeszédablakok egyszerűen modális formok, akkor jó nyomon járunk, de ez még nem tökéletes. Delphiben – és Windowsban is – lehetnek nem modális párbeszédablakaink, és modális formjaink is. Két dolgot kell megismernünk:

- A form kerete, és a felhasználói felülete határozza meg, hogy párbeszédablaknak néz-e ki.
- A második form viselkedését – modális vagy nem modális voltát – a megjelenítésére használt függvény határozza meg (Show vagy ShowModal).

Ennek a két elemnek a kombinálásával bármilyen másodlagos formot készíthetünk, és a többi ide tartozó téma tárgyalása előtt ezzel foglalkozunk.

Második form hozzáadása a programunkhoz

Most már készen állunk egy példára. A nem modális formok megismeréséhez egy második formot adunk az alkalmazásunkhoz – modális formok készítését a párbeszédablakoknál láthatjuk. A Phones példa nagyon egyszerű: a főform (Form1) a telefonszámok, a másodlagos form (Form2) pedig az e-mail címek listáját tartalmazza. Természetesen ehhez a példához kitalált neveket, számokat, és e-mail címeket használtunk, tehát bármilyen kapcsolat létező személyekkel csak a véletlen műve.

Tipp

Új formot alkalmazásunkhoz a Delphi eszköztár **New Form** gombjával, vagy a **File** menü **New Form** pontjával adhatunk. Másik megoldásként kiválaszthatjuk a **File** menü **New** pontját, átmehetünk a **Forms or Dialogs** oldalra, és választhatunk a felkínált form sablonok és varázslók közül. A fejezet legtöbb példájában üres főformmal, és egy vagy több üres másodlagos formmal kezdünk.

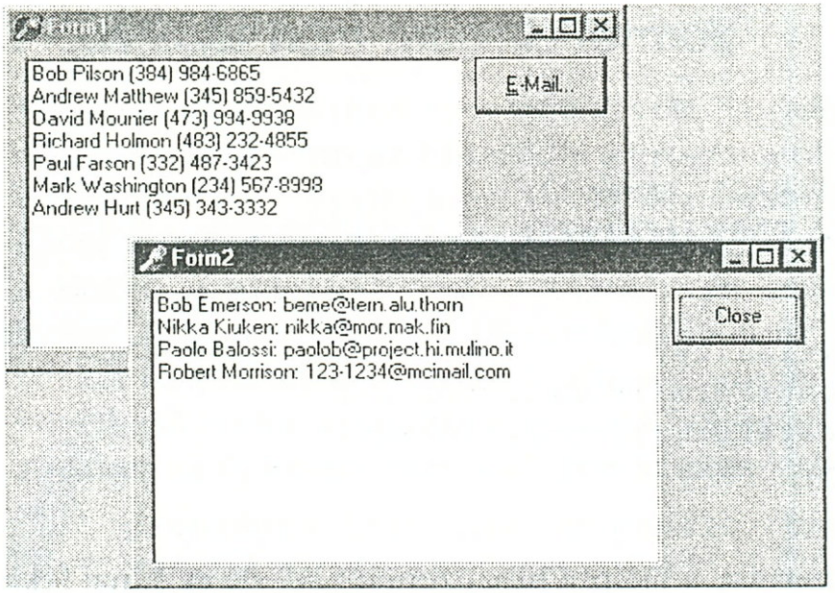
A két form a 13.1. ábrán látható. Ha a projektünkben két form van, akkor szerkesztés közben az eszköztár **Select Form** (form kiválasztása) és **Select Unit** (egység kiválasztása) gombjaival választhatunk közöttük. A Project Options párbeszédablak **Forms** oldalán pedig beállíthatjuk, hogy melyik form legyen a fő, és hogy melyeket kell a program indításakor automatikusan létrehozni – ez a 13.3. ábrán látható.

Miután elkészítettük a két formot, még el kell készítenünk a program kódját a helyes működéshez. Mind a két formon szerepel egy gomb; az első ablakon ez megmutatja a másodikat, a másodikon pedig bezárja a második ablakot. Lehetőségünk van arra, hogy a második ablakot becsukjuk, és az elsőn folytassuk, de ez fordítva nem lehetséges, mert egy Delphi alkalmazás főablakának bezárásakor a program futása megszakad.

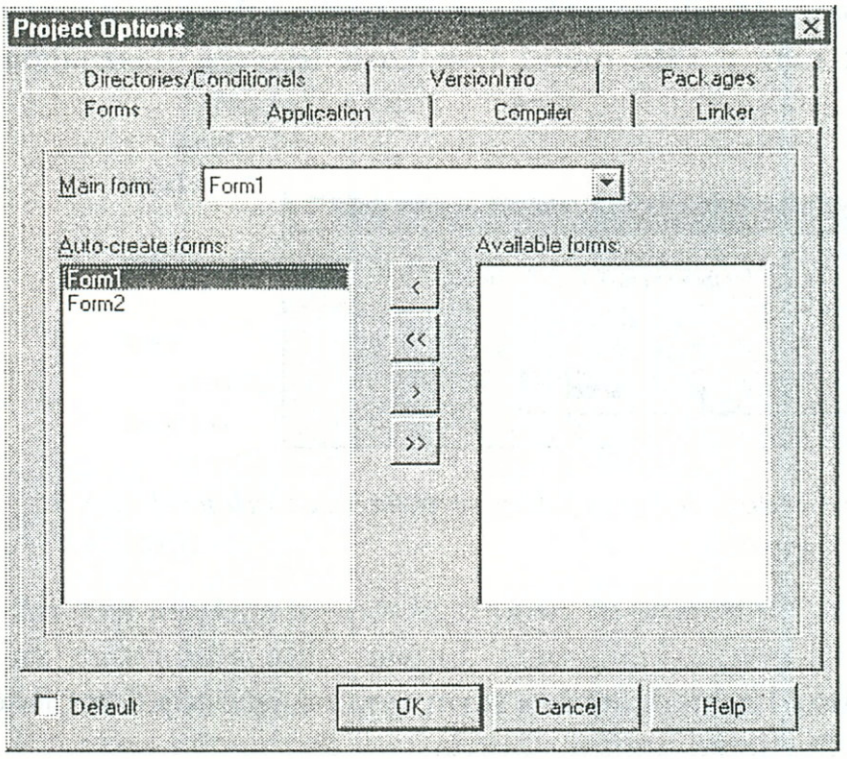
Hogy a Button1 megnyomásakor a második form megjelenjen, ezt az egyszerű kódot használhatjuk:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Show;
end;
```

A Show függvény használatával a második form nem modális ablakként jelenik meg, vagyis visszamehetünk az elsőre mialatt a második még látható. Észrevehettük, hogy a formot csak megjelenítjük, nem hozzuk létre. Ezt azért



13.1. ábra *A Phones példa két formja futás közben*

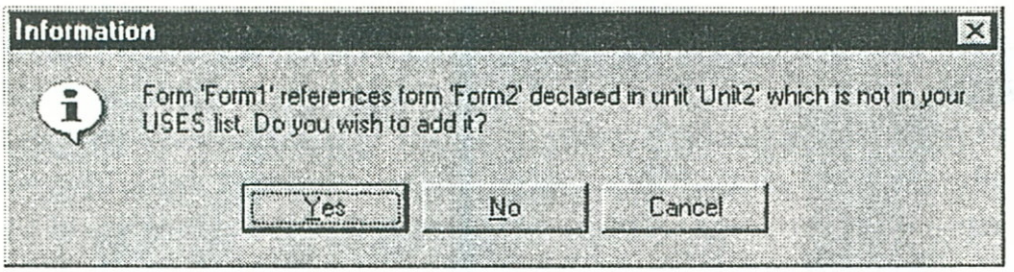


13.2. ábra *A Phones projekt Forms opciói*

tehetjük meg, mert a form még a projekt állományban került létrehozásra, ahogy azt a Project Options párbeszédablak **Forms** oldalán láthattuk. Ha megnézzük a projekt állomány kódját, akkor láthatjuk, hogy az mind a két formot használja, és létre is hozza az alkalmazás CreateForm metódusával. Az elsőként létrehozott form lesz az alkalmazás főformja:

```
program Phones;

uses
  Forms,
  Unit1 in 'UNIT1.PAS' {Form1},
  Unit2 in 'UNIT2.PAS' {Form2};

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

Ha most lefordítjuk az alkalmazást, a Delphi hibaüzenetet ad, de azonnal fel is ajánlja, hogy kijavítja azt, ahogy azt a 13.3. ábrán láthattuk. Az első form kódjának lefordításához használnunk kell a második formot tartalmazó egységet – esetünkben ez a Unit2 – a `uses` kifejezésben, lehetőség szerint az `implementation` részben. Másik lehetőségként kiválaszthatjuk az első formot, és alkalmazhatjuk rajta a **File** menü **Use Unit** (egység használata) parancsát.



13.3. ábra *A fordító rákérdez, hogy a felvegye-e az uses kifejezésébe a hiányzó bejegyzést*

A második form bezárásához használhatjuk a form rendszermenüjét, a Close (bezárás) gombját, ami a következő metódust hívja:

```
procedure TForm2.CloseButtonClick(Sender: TObject);
begin
  Close;
end;
```

Ha ezt a kódot lefuttatjuk, a Delphi nem zárja be a másodlagos formot, csak elrejti, de nekünk erre van szükségünk, hiszen máskor is meg kell tudnunk jeleníteni anélkül, hogy újra elkészítenénk. Más esetekben felüldefiniálhatjuk az OnClose (bezáráskor) eseményt, és ténylegesen megsemmisíthetjük a formot, ezzel elkerülve, hogy túl sok form foglalja a memóriát egy időben.

Modális és nem modális formok

Ahogy az az előző részben is látható volt, nagyon egyszerű második – harmadik, és további – formot hozzáadni a programunkhoz, ezért próbáljunk valami bonyolultabbat. Tegyük a példánkat általánosabbá, amiben a felhasználónak lehetősége van arra, hogy megnyisson több modális vagy nem modális formot. A Modes program fő formján két gomb van, amikkel modális és nem modális formokat hozhatunk létre. Miután hozzáadtunk két új formot a projekthez – amiket mi ModalForm-nak és ModelessForm-nak neveztünk –, megírhatjuk a következő kódot, ami a két gomb OnClick eseménykezelőjének felel meg, és amikben az új formot menet közben hozzuk létre:

```
procedure TMainAppForm.ModalButtonClick(Sender:TObject);
var
  Modal: TModalForm;
begin
  Modal := TModalForm.Create (Application);
  try
    Modal.ShowModal;
  finally
    Modal.Free;
  end;
end;

procedure TMainAppForm.ModelessButtonClick(Sender:
                                                TObject);
var
  NonModal: TModelessForm;
begin
  NonModal := TModelessForm.Create (Application);
  NonModal.Show;
end;
```

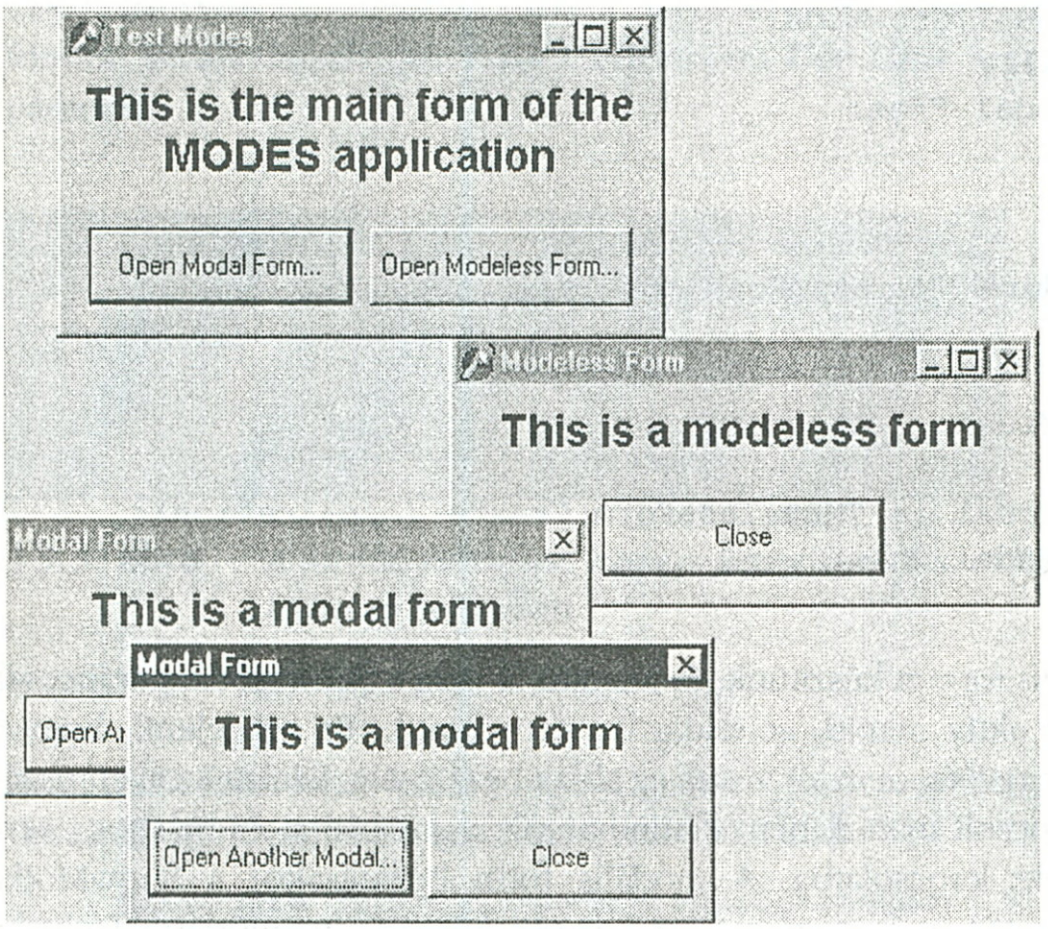
Amikor modális formot készítünk, és jelenítünk meg a ShowModal paranccsal, az mindaddig aktív marad, ameddig be nem zárjuk. Ez azt jelenti, hogy a ShowModal függvény nem tér vissza az ablak bezárásáig. Ezalatt az alkalmazás főformja kikapcsolt marad, nem kattinthatunk a gombjaira, és máshogy sem léphetünk vele kapcsolatba. A modális form bezárásakor a ShowModal függvény visszatér, és a program kitörli az objektumot a memóriából (mivel a ShowModal hívás is okozhat kivételt, ezért, hogy a felszabadítás mindenképpen megtörténjen, azt a finally blokkba írjuk).

A nem modális formok viselkedése ettől eltérő; a Show eljárás – a ShowModaltól eltérően, ami függvény volt, ez eljárás – azonnal visszatér. Természetesen ilyenkor nem semmisíthetjük meg az objektumot, hiszen az ablak éppen a képernyőn van, ezért a form bezárásakor magát szabadítja fel. Megnyomhatjuk azonban a Modeless gombot újra, hogy új ablakokat hozzunk

létre, kettőt, hármat, sőt még többet. Gyakorlatilag nincsen felső határ, a Windows erőforrásaitól eltekintve, amik a 32 bites Windows változatokban igen magasak.

A főform megjeleníthet másodlagos formokat, de mit tehetünk ezekkel a formokkal? Amint az a 13.4. ábrán látható, mindegyiknek van egy Close (Bezárás) gombja. A két Close gomb egy egyszerű metódushoz van kötve, ami a Close eljárást hívja. A modális formokon is van egy gomb, ami egy másik modális formot készít, ennek a gombnak az OnClick metódusának kódja megegyezik a főform ModalButtonClick metódusával.

A program fontos része, hogy mivel TForm osztálybeli objektumokat hozunk létre, ezért a Project Options párbeszédablak **Forms** oldalán – ez már látható volt a 13.2. ábrán – kapcsoljuk ki az automatikus készítést. Szintén távolítsuk el annak a globális változónak a deklarációját, amit a Delphi a formnak automatikusan hozzáad az egység interfész részében a formot leíró részhez.



13.4. ábra *A Modes program modális és nem modális formjai futásidőben*

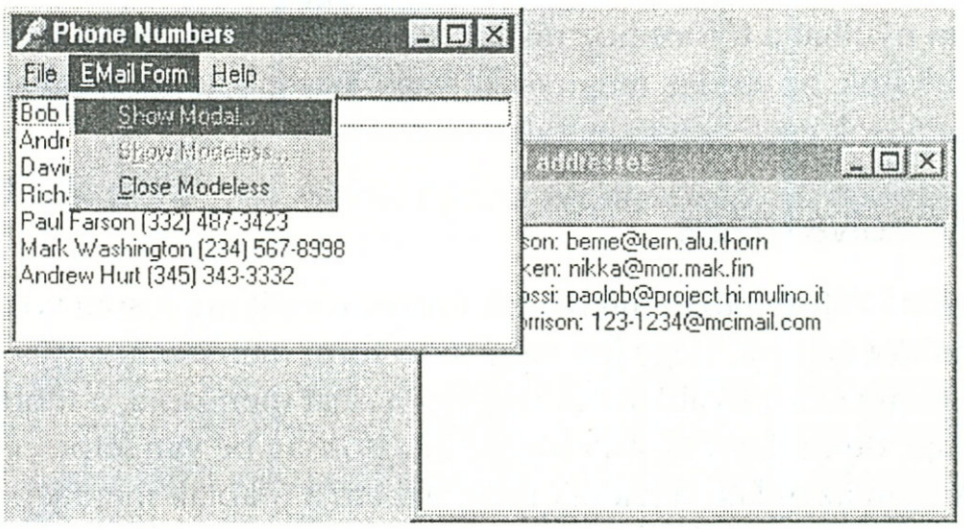
Két form, két menü

Most már visszatérhetünk a Phones példához, hogy menüket adjunk a formokhoz, és ezzel elkészítsük a Phones2 programot. A legegyszerűbb megoldás, ha mind a két formhoz hozzáadunk egy menüt. Hogy bemutassuk, hogy ez a megoldás működik modális és nem modális formokkal egyaránt, ezért a program régebbi változatához hozzáadtuk az a kódot, ami a másodlagos formot modálisan, vagy nem modálisan jeleníti meg, az EMail Form legördülő menü pontjainak megfelelően.

A másodlagos form megjelenítéséhez csak a ShowModal, vagy Show metódusokat kell használnunk. A valóságban mindkét form elkészül a projekt indításakor. Ha a másodlagos form modálisan jelenik meg, a főform – menüjével együtt – automatikusan elveszti a fókuszt, ha viszont a második form nem modális, a menüt célszerű kézzel letiltanunk, hiszen, mivel csak egy másodlagos formunk van, ezt kétszer megjeleníteni nem lenne szerencsés.

```
procedure TForm1.ShowModeless1Click(Sender: TObject);
begin
  ShowModal1.Enabled := False;
  ShowModeless1.Enabled := False;
  Form1.CloseModeless1.Enabled := True;
  Form2.Show;
end;
```

Amikor a form megjelenik, az EMail Form menü első két pontját ki-, a harmadikat pedig bekapcsoljuk – ez utóbbi tervezés alatt volt kikapcsolt –, ahogy az a 13.5. ábrán látható.



13.5. ábra *A Phones2 példa főformja kikapcsolt menüpontokkal, amikor a másodlagos form nem modálisként jelenik meg*

Vegyük észre, hogy a menüpontoknak azért van különböző színük, mert, bár kikapcsolt, az első pont van kiválasztva. Amikor az utolsó menüpontot kiválasztjuk, a nem modális formot bezárjuk a Close metódusának meghívásával. Mi történik a menüpontokkal? Nem kellene visszakapcsolnunk a Close Modeless (nem modális form bezárása) menüpont meghívásakor? Megtehetnénk, de akkor a program nem működne helyesen, hiszen a valóságban több módszer is van a második form bezárására, például a saját menüjének Close parancsa. A kód ismételgetése helyett a menüpontok visszaállítását a második form OnClose eseményében tesszük meg:

```
procedure TForm2.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Form1.ShowModal1.Enabled := True;
  Form1.ShowModeless1.Enabled := True;
  Form1.CloseModeless1.Enabled := False;
end;
```

A második form hivatkozik az elsőre, ezért el kell érnie a TForm1 osztály néhány tulajdonságát. Emiatt a második formnak használnia kell az elsőt, de itt az ezt szolgáló uses kifejezésnek az egység kifejtés (implementation) részében kell szerepelnie. Ha mind a két egység az interfész részben hivatkozna egymásra, akkor körkörös hivatkozás alakulna ki. Ha a két egység hivatkozik egymásra, de a `uses` kifejezéseik a kifejtés részben vannak, akkor nincsen probléma, mert a második egységre már csak akkor lesz szükség, miután az első lefordult.

Ez a példa bemutatta, hogy a modális formok kezelése egyszerűbb a nem modálisakénál, mert ezeknél nem kell megváltoztatnunk a főablak beállításait. Mivel a felhasználó nem nyúlhat a főformhoz mialatt a modális ablak nyitva van, ezért nem kell törődnünk az eddig megnyitott vagy becsukott formokkal, ahogy azt a nem modálisok esetében láthattuk.

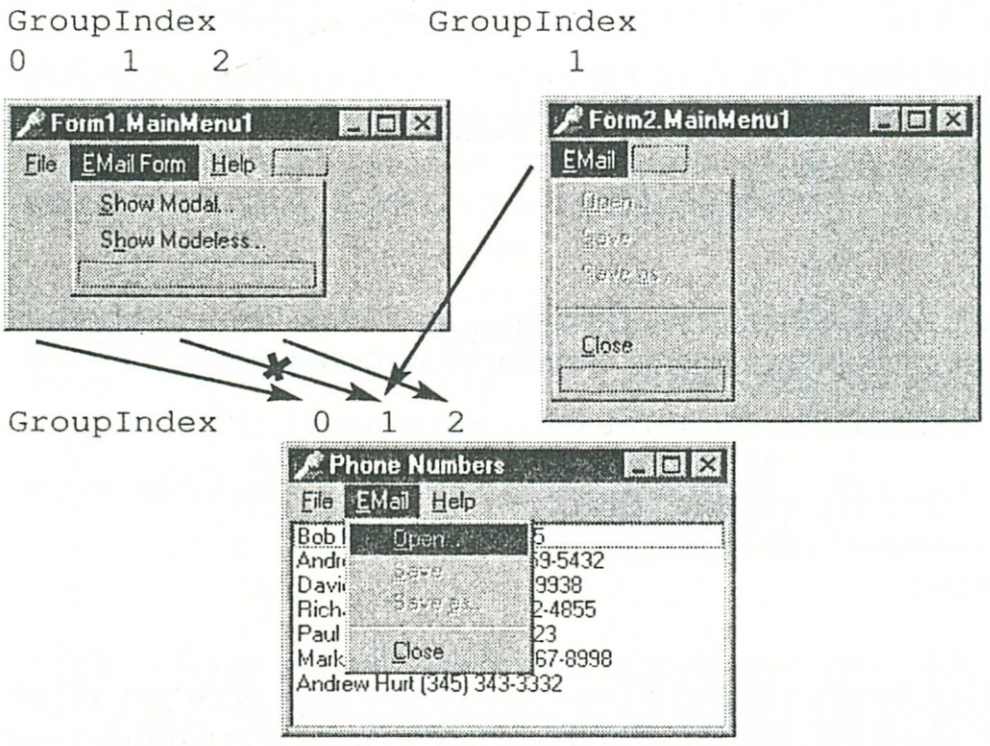
Form menük egybeolvasztása

A Phones program előző változatában mind a két formnak saját menüsora volt. A Delphi azonban módot ad arra, hogy két vagy több form menüsorát egybeolvasszuk. Ezzel a technikával a fő ablaknak megvan a saját menüsora, a többi formnak is van menüje, de ezeknél az AutoMerge tulajdonság be van kapcsolva, így a menüsoruk nem az ablakon jelenik meg, hanem a főablak menüjébe olvad be.

Az egybeolvasztás szabályai a következők: minden legördülő menünek megvan a saját GroupIndex (csoportindex) tulajdonsága. Amikor a menük egybeolvadnak, a legördülő menük sorrendje a következőképp alakul:

- Ha a két különböző menüsorban van azonos GroupIndex-ű elem, akkor az eredeti menü megfelelő pontja eltűnik.
- Az elemek a GroupIndex tulajdonság növekvő sorrendjében szerepelnek.

A 13.6. ábrán láthatjuk a Phones3 példa menüinek összeolvadását. A főform menüinek indexei rendre 0, 1 és 2. A másodlagos form egyetlen legördülő menüjének 1-es indexe van, ezért ez kicseréli a főmenü ugyanilyen indexű menüpontját. Ne felejtsük el, hogy a menük egybeolvasztásának csak nem modális formok esetében van értelme. Ha a másodlagos formot modálisan jelenítjük meg, akkor a fő ablak menüjét nem érhetjük el, ezért a Delphi nem is olvaszt össze semmit.



13.6. ábra *A Phones3 példa két menüsora, és összekapcsolódásuk*

A menük egybeolvasztása, első ránézésre legalábbis, nagyon egyszerű. Ha újra megnyitjuk a Phones2 példát, a második form főmenüjének AutoMerge tulajdonságát igazra állítjuk, és a GroupIndex tulajdonságnak a következő listán látható értékeket adjuk, akkor a program majdnem működőképes lesz:

Form	Legördülõ menü	GroupIndex
Form1	File	0
	EMail Form	1
	Help	2
Form2	EMail	1

Van azonban még egy probléma, miszerint, ha a második form modálisan jelenik meg, ezért a menü nem elérhető. Ezt a problémát egyszerűen megoldhatjuk, ha a második form menüjének AutoMerge tulajdonságát, programból, dinamikusan állítjuk.

A sima tulajdonságállítás helyett itt egy sokkal radikálisabb, dinamikusabb megközelítést fogunk alkalmazni, hogy bemutassunk egy újabb példát a dinamikus form létrehozásra. A másodlagos formot minden megjelenésekor létrehozzuk, és elrejtésekor megsemmisítjük. Ehhez több apró változtatást kell végrehajtanunk a forráson. Először ki kell kapcsolni az automatikus form készítést a Project Options párbeszédablak **Forms** oldalán. Ahhoz, hogy a formunk modálisan jelenjen meg, először létre kell hoznunk, be kell állítanunk a tulajdonságait, és végül meg kell semmisítenünk:

```
procedure TForm1.ShowModal1Click(Sender: TObject);
begin
  Form2 := TForm2.Create (Self);
  Form2.MainMenu1.AutoMerge := False;
  Form2.ShowModal;
  Form2.Free;
end;
```

A nem modális esetben alkalmazott megközelítés ettől csak kismértékben tér el, itt ugyanis a formot csak létrehozzuk, de nem semmisítjük meg:

```
procedure TForm1.ShowModeless1Click(Sender: TObject);
begin
  Form2 := TForm2.Create (Self);
  Form2.MainMenu1.AutoMerge := True;
  Form2.Show;
end;
```

Észrevehettük, hogy a menüpontokat kikapcsoló részlet nem szerepel a kódban, hiszen az egész legördülő menü eltűnik. Szintén emiatt távolítottuk el az utolsó pontot, a Close-t. A nem modális form megsemmisül bezáráskor, ha az OnClose eseményében az Action paraméter értékét caFree-re állítjuk:

```
procedure TForm2.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;
end;
```

A Phones3 példa hátralévő része megegyezik a Phones2-vel, bár az utolsó módosítások után nem maradt túl sok minden érintetlenül.

Párbeszédablakok készítése

Ahogy az ebben a fejezetben korábban elmondtuk, a párbeszédablakok nem különböznek lényegesen a másfajta formoktól. Egy egyszerű trükkel formok helyett párbeszédablakokat készíthetünk, elég, ha a form BorderStyle (keret stílus) tulajdonságánál bsDialog értéket állítunk be. Ennek az egyszerű változtatásnak a hatására a form felülete egy párbeszédablakéhoz fog hasonlítani, vagyis nem lesz rendszer menü ikonja, kis-, vagy teljes méret ikonja, és ha a címsorán jobb gombbal kattintunk, akkor az ilyenkor megfelelő rendszermenü jelenik meg. Természetesen, az ilyen formoknak a párbeszédablakoknál megszokott vékony, nem átméretezhető kerete van. A keretstílusokat a 10. fejezet Borders példájában láthattuk.

Miután elkészítettük a párbeszédablakunk formját, akkor azt a már ismert két metódussal megjeleníthetjük akár modálisan (a ShowModal függvénnyel), akár nem modálisan (a Show eljárással). A párbeszédablakok általában modálisak. Ebben térnek el a formoktól: általában kerüljük a modális formok használatát, mert a felhasználók nem számítanak rá. A következő táblázat tartalmazza a különböző stílus változatok ismertetését:

Ablak típus	Modális	Nem modális
Form	soha	SDI alkalmazásoknál általában
Párbeszédablak	A másodlagos formok legáltalánosabb fajtája	Ritka, de használt

Néha a túl sok másodlagos formot elkerülendő több oldalas formokat készítenek, ami a Windows alkalmazások között nagyon gyorsan terjedő új felhasználói felület. Ezt mi is könnyen megtehetjük a Delphi Notebook (jegyzettömb) és PageControl (füles kontrol) komponenseivel, ahogy azt a tizennegyedik fejezetben láthatjuk.

Modális párbeszédablakok

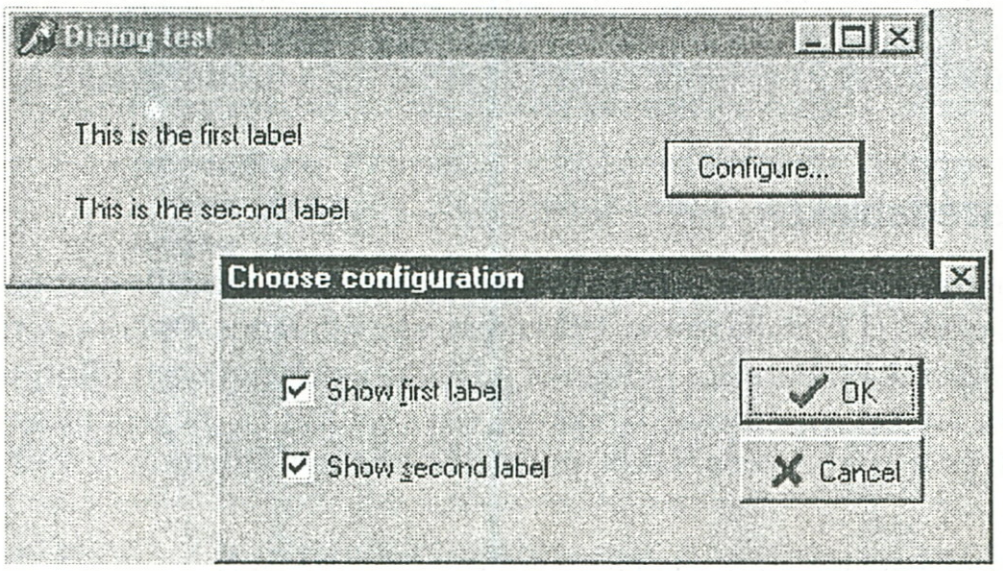
Mivel a modális párbeszédablakok sokkal elterjedtebbek nem modális társaiknál, ezért először ezekkel foglalkozunk. Példaként egy általános Opciók párbeszédablakot készítünk el. Ennek a példának a Dialog1-nek nevezett főformján két címke van, amiket a párbeszédablakból elrejthetünk és megjeleníthetünk. A főformot és a párbeszédablakot működés közben láthatjuk a 13.7. ábrán.

A párbeszédablakon van még két általános képes nyomógomb, amiknek az alapértelmezett képei OK (Rendben) és Cancel (Mégsem). Általában jó ötlet

ezeket az alapértelmezett gombokat használni, mert elég csak egyetlen tulajdonságukat (Kind – fajta) állítani, nem kell külön értéket adni a Caption, a Glyph tulajdonságoknak, a visszatérési értéknek (ModalResult), és még másoknak. Ennek következtében – és ez nem elhanyagolható – általános felhasználói felületet kapunk. Itt következnek a párbeszédablak formjának, és komponenseinek tulajdonságai:

```
object ConfigureDialog: TConfigureDialog
  BorderStyle = bsDialog
  Caption = 'Choose configuration'
  object BitBtn1: TBitBtn
    Kind = bkOK
  end
  object BitBtn2: TBitBtn
    Kind = bkCancel
  end
  object CheckBox1: TCheckBox
    Caption = 'Show &first label'
    State = cbChecked
  end
  object CheckBox2: TCheckBox
    Caption = 'Show &second label'
    State = cbChecked
  end
end
```

Az alkalmazás kódja nagyon egyszerű. Miután a főform kódjába beszerkesztettük a párbeszédablakot leíró formot tartalmazó egységet, meghívjuk a ShowModal függvényt, hogy megjelenítsük a párbeszédablakot. Ez eddig még nem túl hasznos, hiszen nem válaszolunk a felhasználónak a párbeszédablakon végzett ténykedéseire.



13.7. ábra *A Dialog1 példa főformja és a párbeszédablak*

A párbeszédablak futtatása után a két jelölőnégyzet állapota alapján állíthatjuk be a címkék láthatóságát. Azonban, ha a felhasználó a Cancel gomb megnyomásával zárta be a párbeszédablakot, akkor az értékeket figyelmen kívül kell hagynunk, vagyis az értékadásokat csak akkor kell végrehajtani, ha a ShowModal függvény visszatérési értéke – ez a modális eredmény – mrOK:

```
procedure TForm1.ConfigureButtonClick(Sender: TObject);
begin
  if ConfigureDialog.ShowModal = mrOk then
  begin
    Label1.Visible :=ConfigureDialog.CheckBox1.Checked;
    Label2.Visible :=ConfigureDialog.CheckBox2.Checked;
  end;
end;
```

Ha megnyitjuk a párbeszédablakot, beállítunk néhány dolgot, és megnyomjuk a Cancel gombot, akkor az új értékek nem számítanak, de ha újra megnyitjuk a párbeszédablakot, akkor a Cancel megnyomása előtti értéket fogjuk látni, a valóban beállítottak helyett. Emiatt minden futtatáskor helyesen be kell állítanunk a párbeszédablak értékeit. Íme a Dialog1 példa végleges kódja:

```
procedure TForm1.ConfigureButtonClick(Sender: TObject);
begin
  {copy current values to the dialog box}
  ConfigureDialog.CheckBox1.Checked := Label1.Visible;
  ConfigureDialog.CheckBox2.Checked := Label2.Visible;
  if ConfigureDialog.ShowModal = mrOk then
  begin
    {copy new dialog box values in the main form}
    Label1.Visible :=ConfigureDialog.CheckBox1.Checked;
    Label2.Visible :=ConfigureDialog.CheckBox2.Checked;
  end;
end;
```

Ez az egyik megközelítés, amit modális párbeszédablakok esetében követhetünk, és a következő lépésekből áll:

- A párbeszédablak megjelenítése előtt állítsuk be a kezdőértékeket.
- Jelenítsük meg a párbeszédablakot.
- Amennyiben az OK gombbal fejeződött be a párbeszédablak futása, az új értékeket másoljuk vissza a formra.

Természetesen nem ez az egyetlen lehetséges megoldás. Másik lehetőségként megtehetjük azt, hogy a párbeszédablak értékeit csak egyszer, a legelső alkalommal állítjuk be, az aktuális értékeket minden megjelenítés előtt elmentjük, és ha a felhasználó a Cancel gombot nyomja meg, akkor ezeket visszaállítjuk. Az előző példában ezt is írhattuk volna:

```
procedure TForm1.ConfigureButtonClick(Sender: TObject);
var
  old1, old2: Boolean;
begin
  {store the old values of the dialog box}
  old1 := ConfigureDialog.CheckBox1.Checked;
  old2 := ConfigureDialog.CheckBox2.Checked;
  if (ConfigureDialog.ShowModal = mrOk) then
  begin
    {set the new values in the form}
    Label1.Visible :=ConfigureDialog.CheckBox1.Checked;
    Label2.Visible :=ConfigureDialog.CheckBox2.Checked;
  end
  else
  begin
    {restore the old values of the dialog box}
    ConfigureDialog.CheckBox1.Checked := old1;
    ConfigureDialog.CheckBox2.Checked := old2;
  end;
end;
```

Ennek az az előnye, hogy a régi értékeket elmentő és visszaállító kódrészletet akár a párbeszédablakba is tehetjük, ami akkor hasznos, ha a kódból több helyen is meg akarjuk jeleníteni ugyanazt a párbeszédablakot. Ebben az esetben a párbeszédablak beállítását – és akár még a megjelenítését is – beletehetjük a párbeszédablak egyik saját metódusába, és a párbeszédablak OK gombjának OnClick kezelője végezheti az értékek visszamásolását a főformra. Ez azonban más nézőpontot kíván: ahelyett, hogy a főform számára tennénk elérhetővé a párbeszédablak tulajdonságait, a párbeszédablaknak kell tudnia közvetlenül olvasni és írni a főform értékeit.

Párbeszédablak bezárása

Mi történik, amikor a felhasználó megnyomja az OK vagy a Cancel gombot? A kódnak be kell csuknia a párbeszédablakot, és a megfelelő értéket vissza kell adnia az alkalmazásnak. A visszatérési értéket a form lezárását végző gomb ModalResult (modális eredmény) tulajdonságában állíthatjuk be. Ha a form ezen tulajdonságának értéket adunk, akkor a modális párbeszédablak automatikusan bezárásra kerül.

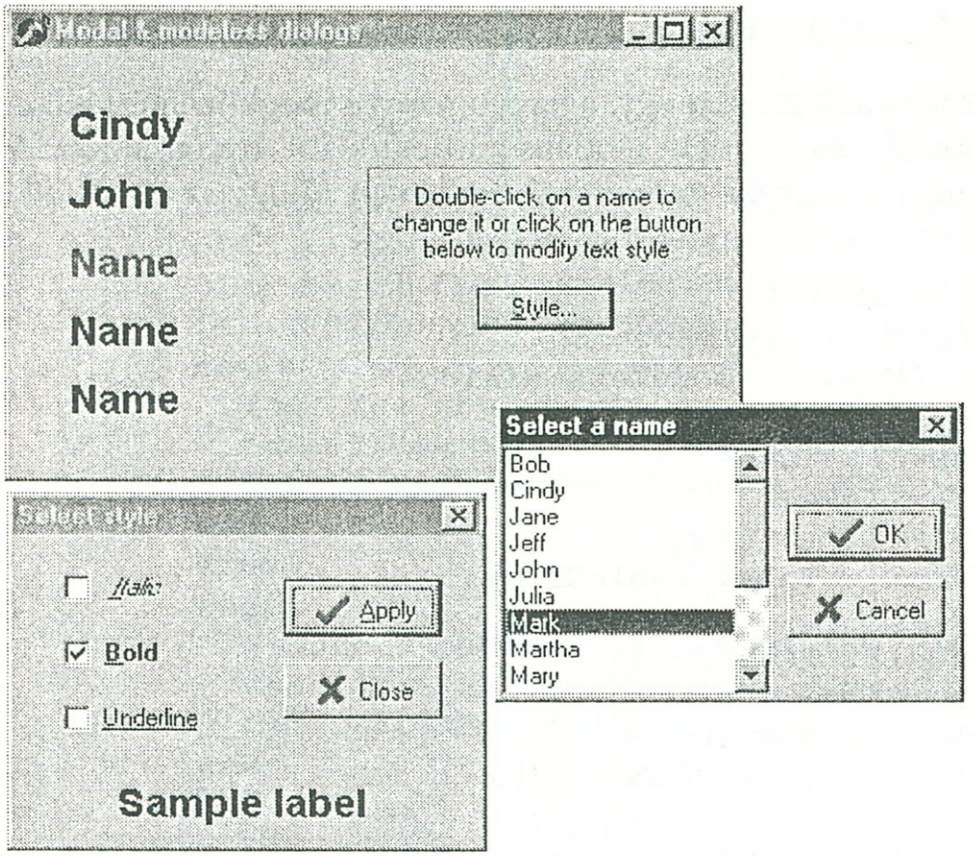
Vegyük észre, hogy a ModalResult értéke lesz az ablak megjelenítésére szolgáló ShowModal függvény visszatérési értéke. Ezért, ezt a kódot nem kell elkészítenünk, mert minden gombnak, a képes gombokat is beleértve, van ModalResult tulajdonsága. Ha egy gomb OnClick eseményét nem kezeljük le, akkor az automatikusan a ModalResult tulajdonságára állítja a szülő form ugyanilyen tulajdonságát.

Nem modális párbeszédablak

A párbeszédablakok második példája egy, a hagyományos megközelítést használó összetettebb modális, és egy nem modális párbeszédablakot tartalmaz. A Dialog2 példa főformján öt, neveket tartalmazó címke van, ahogy az a 13.8. ábrán is látható:

```
object Form1: TForm1
  ActiveControl = StyleButton
  Caption = 'Modal & modeless dialogs'
  object Bevel1: TBevel...
  object Label1: TLabel
    Caption = 'Name'
    OnClick = LabelClick
    OnDblClick = LabelDoubleClick
  end
  object Label2: TLabel
    Caption = 'Name'
    OnClick = LabelClick
    OnDblClick = LabelDoubleClick
  end
  object Label3: TLabel... // the same
  object Label4: TLabel... // the same
  object Label5: TLabel... // the same
  object Label6: TLabel
    Alignment = taCenter
    AutoSize = False
    Caption = 'Double-click on a name to change it
      or click on the button below to modify text
      style'
    WordWrap = True
  end
  object StyleButton: TButton
    Caption = '&Style...'
    OnClick = StyleButtonClick
  end
end
```

Ha a felhasználó egy néven kattint, annak a színe pirosra vált; ha duplán kattint rajta, akkor a program megjelenít egy neveket tartalmazó modális párbeszédablakot, amiből választani lehet. Ha a felhasználó a Style (stílus) gombon kattint, akkor egy nem modális párbeszédablak jelenik meg, amin a főform címkéinek betűtípusát változtathatjuk meg. A főform mind az öt címkéje ugyanazon két metódushoz kötődik, az egyik az OnClick, a másik az OnDoubleClick eseménykezelő. Az első metódus pirosra állítja azt a címkét, amire kattintottunk, a többit pedig visszafeketíti:



13.8. ábra *A Dialog2 példa három formja (a főform, és a két párbeszédablak) futás közben*

```
procedure TForm1.LabelClick(Sender: TObject);
begin
  {set the color of all the labels to black}
  Label1.Font.Color := clBlack;
  Label2.Font.Color := clBlack;
  Label3.Font.Color := clBlack;
  Label4.Font.Color := clBlack;
  Label5.Font.Color := clBlack;
  {set the color of the clicked label to red}
  (Sender as TLabel).Font.Color := clRed;
end;
```

A második metódus az OnDoubleClick eseménykezelő. A LabelDoubleClick metódus a Sender paraméter által jelzett, aktuális címke tartalmát kiválasztja a párbeszédablak listadobozából, és megjeleníti a modális párbeszédablakot. Ha a felhasználó az OK gomb megnyomásával zárja be az ablakot, és a listának van kiválasztott eleme, akkor a címke tartalma arra változik:

```
procedure TForm1.LabelDoubleClick(Sender: TObject);
begin
  with ListDial.Listbox1 do
  begin
```

```
    {select the current name in the list box}
    ItemIndex := Items.IndexOf (Sender as
      TLabel).Caption);
    {show the modal dialog box, checking the return
      value}
    if (ListDial.ShowModal = mrOk) and (ItemIndex >= 0)
    then
      {copy the selected item to the label}
      (Sender as TLabel).Caption := Items [ItemIndex];
end;
```

Figyeljük meg, hogy a LabelDoubleClick metódus tartalmazza a modális párbeszédablak beállításait végző valamennyi kódot. A párbeszédablak formjában nincs hozzáadott kód:

```
object ListDial: TListDial
  BorderStyle = bsDialog
  Caption = 'Select a name'
  object ListBox1: TListBox
    Align = alLeft
    Items.Strings = (
      'Bob'
      'Cindy'
      ...)
    Sorted = True
  end
  object BitBtn1: TBitBtn
    Kind = bkOK
  end
  object BitBtn2: TBitBtn
    Kind = bkCancel
  end
end
```

Tipp

Ha általános kódot írunk olyan metódusban, ami csak egyfajta komponenshez kapcsolódik, akkor nem kell leellenőriznünk, hogy a Sender milyen fajta, olyan kifejezéssel, mint if Sender is TLabel. A valóságban a Sender mindig címke. Ha valamilyen oknál fogva mégsem, akkor hibaüzenetet kapunk a mikor a kifejezésre kerülne a vezérlés, és ez azt jelenti, hogy a feltételezésünk, hogy a metódus mindig azonos típusú komponensekhez kapcsolódik, hamis, és hálásak lehetünk a hibaüzenetnek, és kijavíthatjuk a programot.

A nem modális párbeszédablak esetében azonban rengeteg háttérkódra van szükség. A főform továbbra is fut, miután a Style gombot megnyomjuk – a gomb neve három ponttal végződik, ami azt jelzi, hogy megnyomásakor párbeszédablak jelenik meg –, mert a Show metódust használjuk. A párbeszédablakot működés közben a 13.8. ábrán, a szöveges leírását pedig itt láthatjuk:

```
object StyleDial: TStyleDial
  BorderStyle = bsDialog
  Caption = 'Select style'
  object LabelSample: TLabel
    Alignment = taCenter
    Caption = 'Sample label'
  end
  object ApplyBitBtn: TBitBtn
    Caption = '&Apply'
    ModalResult = 1 // mrOK
    OnClick = ApplyBitBtnClick
  end
  object CloseBitBtn: TBitBtn
    Caption = 'Close'
    ModalResult = 2 // mrCancel
    OnClick = CloseBitBtnClick
  end
  object ItalicCheckBox: TCheckBox
    Caption = '&Italic'
    Font.Style = [fsItalic]
    OnClick = ItalicCheckBoxClick
  end
  object BoldCheckBox: TCheckBox
    Caption = '&Bold'
    Font.Style = [fsBold]
    OnClick = BoldCheckBoxClick
  end
  object UnderlineCheckBox: TCheckBox
    Caption = '&Underline'
    Font.Style = [fsUnderline]
    OnClick = UnderlineCheckBoxClick
  end
end
```

Figyeljük meg az Apply (alkalmaz) és a Close (bezár) gombok neveit, amik általában az Ok és Cancel gombokat helyettesítik nem modális párbeszédablakokon. Ezeket a gombokat legkönnyebben úgy hozhatjuk létre, hogy a Kind tulajdonságukhoz beállítjuk a bkOk és a bkCancel értékeket, és ezután írjuk át a címkéjüket. Néha láthatunk Cancel gombot, ami Close-ként működik, de nem modális párbeszédablak esetében az OK gombnak általában semmi értelme nincs. Ehelyett egy, vagy több olyan gombot célszerű használnunk, amik a

főablakon módosításokat hajtanak végre, ilyen lehet az Alkalmaz, a Stílus Váltása, a Csere, Törlés, és így tovább.

Amikor a felhasználó a nem modális párbeszédablakunk jelölőnégyzetein kattint, az alul látható példacímke szöveg stílusa megváltozik. Ezt a megfelelő stílusokat jelölő jelzők beállításával, vagy törlésével érjük el, ahogy az a következő OnClick eseménykezelőből látszik:

```
procedure TStyleDial.ItalicCheckBoxClick(Sender:
                                                TObject);
begin
  if ItalicCheckBox.Checked then
    LabelSample.Font.Style :=
      LabelSample.Font.Style + [fsItalic]
  else
    LabelSample.Font.Style :=
      LabelSample.Font.Style - [fsItalic];
end;
```

Ha a felhasználó az Apply gombot választja, a program a jelölődobozok állapotának vizsgálata helyett, a példacímke stílusát átmásolja a form címkéibe:

```
procedure TStyleDial.ApplyBitBtnClick(Sender: TObject);
begin
  Form1.Label1.Font.Style := LabelSample.Font.Style;
  Form1.Label2.Font.Style := LabelSample.Font.Style;
  ...
```

Másik megoldásként, közvetlenül a címkékre hivatkozás helyett használhatjuk a form FindComponent metódusát, aminek átadjuk a címke nevét paraméterként, és a végeredményt TLabel típusúként kezeljük. Ennek a megoldásnak az előnye, hogy a címkeneveket egy for ciklussal állíthatjuk elő:

```
procedure TStyleDial.ApplyBitBtnClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 5 do
    (Form1.FindComponent ('Label' + IntToStr (I)) as
      TLabel).
        Font.Style := LabelSample.Font.Style;
end;
```

Tipp

Az ApplyBitBtnClick metódust úgy is megírhatjuk, hogy az a Controls tömböt egy ciklusban nézi végig, ahogy azt más példákban tettük. Azért döntöttünk a FindControl metódus használata mellett, hogy bemutassuk ezt a lehetőséget is.

Ez a második kódrészlet egyértelműen lassabb, mert több dolgot kell tennie, de észre sem fogjuk venni a különbséget, mert még így is nagyon gyors. Természetesen, ez a második megoldás általánosabb, mert ha új címkét adunk a formhoz, akkor csak a ciklus felsőértékét kell a címkék számának megfelelően módosítani. Vegyük észre, hogy amikor a felhasználó az Apply gombra kattint, a párbeszédablak nem záródik be, ezt csak a Close gombbal tehetjük meg. Azt is érdemes megfigyelni, hogy a párbeszédablakot nem kell inicializálnunk, mert a form nem semmisül meg, és a komponensei is megtartják a státuszukat amikor a párbeszédablak újra megjelenik.

Előre definiált párbeszédablakok használata

Saját párbeszédablakok készítése mellett a Delphi rendelkezik néhány alapvető párbeszédablakkal. Néhányat ezek közül, mint például az üzenet ablakokat, a Windows definiál, és a Delphi csak megjelenít, de a Delphiben vannak olyan form sablonok és varázslók, amikkel szintén készíthetünk párbeszédablakokat, ahogy azt a harmadik fejezetben megnéztük.

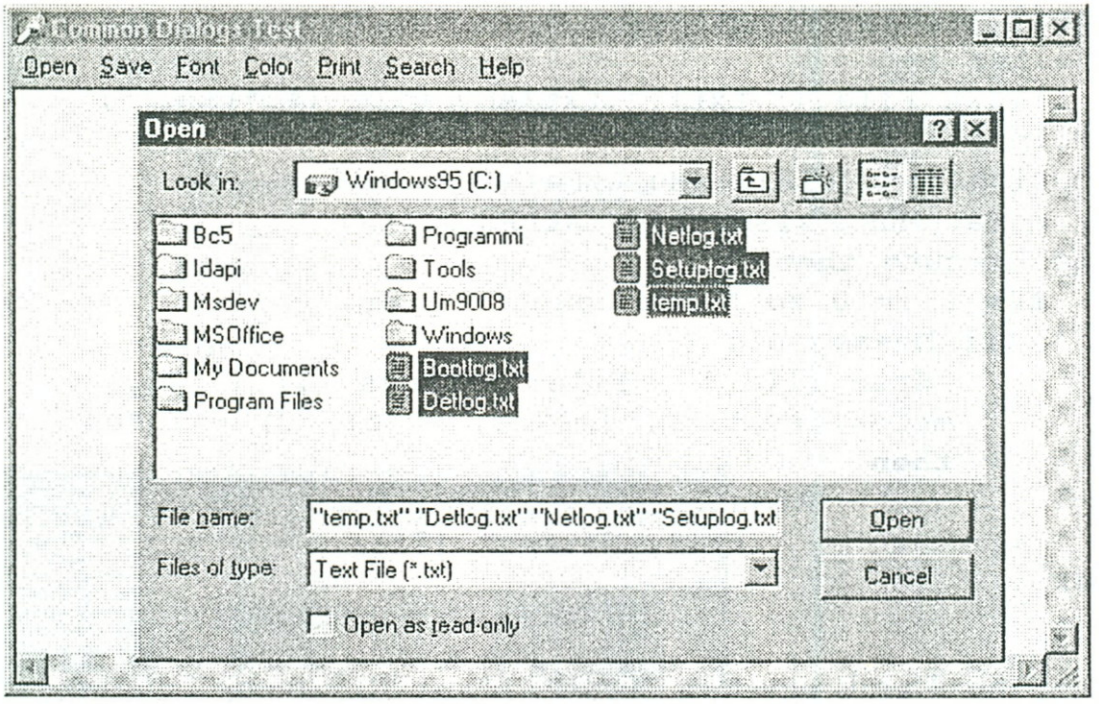
Általános Windows párbeszédablakok

A Delphi komponens palettájának van egy oldala, ami párbeszédablak komponenseket tartalmaz. Ezeket a általános Windows párbeszédablakoknak is nevezett párbeszédablakokat a COMMDLG.DLL definiálja, ami először a Windows 3.1-ben jelent meg. Ezek közül néhányat már használtunk az előző fejezetek példáiban, így ezeket már ismerhetjük. Elég egyszerűen feltennünk a megfelelő komponenst a formra, beállítani néhány tulajdonságát, és az Execute metódussal futtathatjuk is a párbeszédablakot, majd lekérdezhetjük a futás eredményeként beállított tulajdonságokat.

Az Open Dialog komponens

Ebben a részben egy tesztprogramot készítünk, a CommDlg-t, hogy bemutassuk az általános párbeszédablakok néhány jellemzőjét. A valóságban néhány opció és tulajdonság beállításával ugyanazon párbeszédablak nagyon eltérő változatait kaphatjuk, a viselkedését és kinézetét jobban megváltoztathatjuk, mint az gondolnánk. Például az OpenDialog (állomány megnyitási párbeszédablak) komponensnél szűrőket állíthatunk be, ezzel ellenőrizve, hogy a kiválasztott állomány kiterjesztése megfelel-e a beállított kiterjesztésnek – erre az Options tulajdonság ofExtensionDifferent (eltérő kiterjesztés) jelzőjét használhatjuk –, vagy engedélyezhetjük több állomány kijelölését is.

A következő két metódus közül az egyik olyan párbeszédablakot állít elő, ami egy meghatározott kiterjesztést ellenőriz, a másik pedig lehetővé teszi több állomány kijelölését – ez a 13.9.ábrán is látszik:



13.9. ábra *Az OpenDialog komponens ofAllowMultiSelect opciója lehetővé teszi több állomány kiválasztását, ahogy az a CommDlg példán látható*

```
procedure TCommDlgForm.TextFiles1Click(Sender: TObject);
begin
  with OpenDialog1 do
  begin
    Filter := 'Text File (*.txt)|*.txt';
    DefaultExt := 'txt';
    Filename := '';
    Options := [ofHideReadOnly, ofFileMustExist,
      ofPathMustExist];
    if Execute then
      if ofExtensionDifferent in Options then
        MessageDlg (
          'Not a file with the .TXT extension',
          mtError, [mbOK], 0)
      else
        Memo1.Lines.LoadFromFile (FileName);
  end;
end;

procedure TCommDlgForm.MultipleSelection1Click(Sender:
TObject);
var
  I: Integer;
begin
  with OpenDialog1 do
```

```
begin
  Filter :=
    'Text File (*.txt)|*.txt|Any File (*.*)|*.*';
  Filename := '';
  Options := [ofAllowMultiSelect,
    ofPathMustExist, ofCreatePrompt];
  if Execute then
    for I := 0 to Files.Count - 1 do
      if MessageDlg (
        'Open file ' + Files.Strings [I] + '?',
        mtConfirmation, [mbYes, mbNo], 0) = idYes
        then
              Memo1.Lines.LoadFromFile
                (Files.Strings [I]);
end;
end;
```

Természetesen, mivel csak egyetlen Memo-t használunk megjelenítésre, az utolsó művelet végén csak az utolsónak megnyitott állomány tartalmát fogjuk látni. A SaveDialog komponenst is így használhatjuk.

A Font Dialog komponens

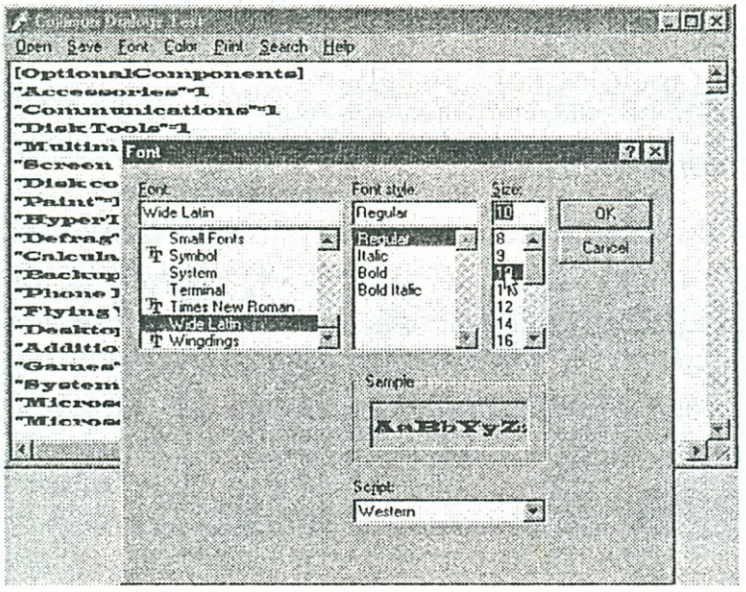
A FontDialog komponenst négyféle változatban jeleníthetjük meg. A Font legördülő menü első pontja csak TrueType betűtípusokat jeleníti meg, és tartalmazza a különleges beállítások részét is. A második menüpontban bármilyen betűtípust kiválaszthatunk, de a különleges hatásokat nem érhetjük el, ez látható a 13.10. ábrán. A harmadik változatban a stílusok és a méret sem állítható.

A három párbeszédablak megjelenítéséhez használt kód csak a futtatás előtt az Options tulajdonságra beállított értékben tér el, ezek sorra:

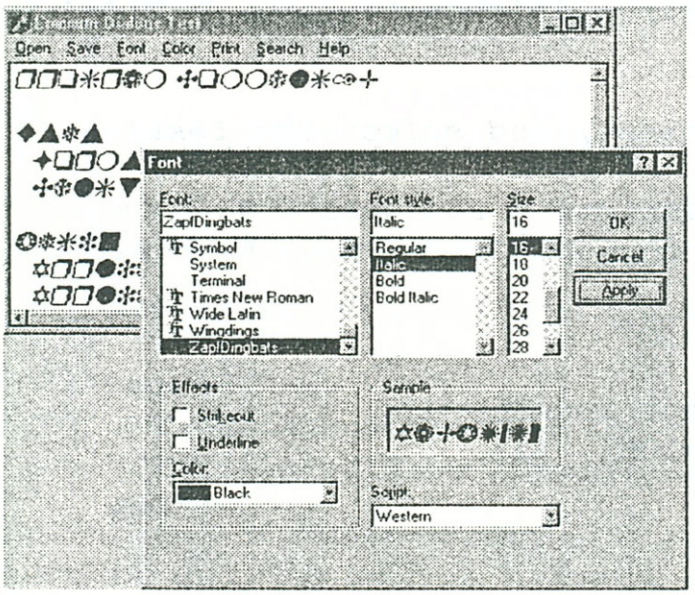
```
Options := [fdEffects, fdTrueTypeOnly,
fdForceFontExist];
Options := [fdForceFontExist];
Options := [fdEffects, fdNoOEMFonts, fdNoStyleSel,
  fdNoSizeSel, fdForceFontExist];
```

A betűtípus választó párbeszédablak utolsó változata még érdekesebb. Ebben a változatban bekapcsoltuk a párbeszédablak Apply (alkalmaz) gombját a fdApplyButton opcióval, és azzal, hogy az OnApply eseményéhez kezelőt írtunk. A metódus végén ezt a kezelőt el kell távolítanunk, az érték nil-re állításával, hogy a program ugyanezt a párbeszédablakot használhassa enélkül is. A betűtípus választó párbeszédablak, lásd a 13.11. ábrát, Apply gombja majdnem úgy működik, mint egy nem modális párbeszédablak, bár nem az:

```
procedure TCommDlgForm.Apply1Click(Sender: TObject);
begin
  with FontDialog1 do
  begin
    OnApply:=FontDialogApply;
    Options:=[fdEffects,fdForceFontExist,fdApplyButton];
    Execute;
    OnApply:=nil;
  end;
end;
```



13.10. ábra *A Font párbeszédablak a beállítások rész nélkül, egyike azoknak a változatoknak, amiket a CommDlg példa bemutat*



13.11. ábra *A Font kiválasztó párbeszédablak az Apply gombbal*

Amikor a felhasználó megnyomja az Apply gombot, a hozzárendelt eseménykezelő lefut. Ez a Memo1 komponens Font tulajdonságába másolja a párbeszédablak beállításait. Ne felejtsük el, hogy a gomb a párbeszédablak része, de az eseménykezelő a program főformjának része. Ez nagyon hasznos, mert a párbeszédablak viselkedését megváltoztathatjuk anélkül, hogy a komponenshez kellene nyúlnunk.

A Color Dialog komponens

A ColorDialog komponenst több opcióval használhatjuk; azonnal megmutathatjuk a teljes párbeszédablakot, vagy megtilthatjuk a teljese megjelenítését. Ezekre szolgál a cdFullOpen (teljes megjelenítés) vagy a cdPreventFullOpen (teljes megjelenítés tiltása) érték az Opciók tulajdonságban.

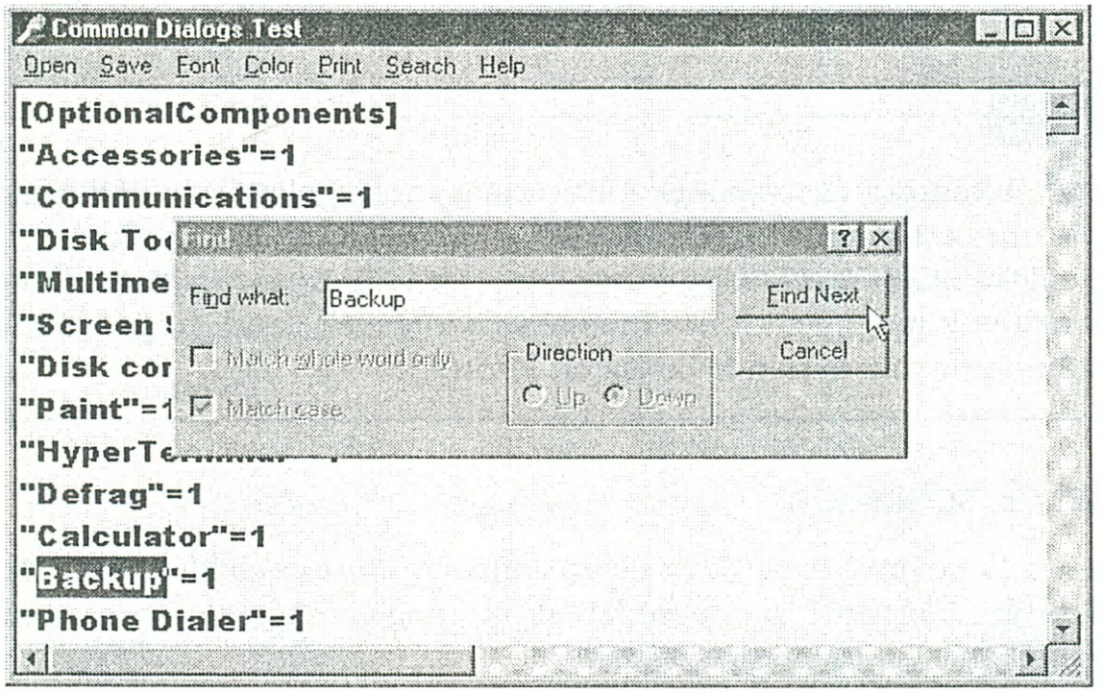
A Find Dialog és Replace Dialog komponensek

A Find és a Replace párbeszédablakok igazi nem modális párbeszédablakok. A Find és Replace metódusoknak csak egy részét készítettük el, és hozzákötöttük a két párbeszédablak gombjához, ahogy azt a betűtípus választó ablak esetében is tettük. Ez a 13.12.ábrán látható. A FindDialog1Find metódus a Pos rendszer függvény segítségével egyezést keres a memo szövegében, illetve az aktuális kiválasztást követő részén:

```
procedure TCommDlgForm.FindDialog1Find(Sender: TObject);
var
  FoundPos, InitPos: Integer;
begin
  {search in the text after the current selection}
  InitPos := Memo1.SelStart + Memo1.SelLength;
  FoundPos := Pos (FindDialog1.FindText,
    Copy (Memo1.Text, InitPos + 1,
      Length (Memo1.Text) - InitPos));
  if FoundPos > 0 then
  begin
    {activate the component, and select the text}
    Memo1.SetFocus;
    Memo1.SelStart := InitPos + FoundPos - 1;
    Memo1.SelLength := Length (FindDialog1.FindText);
  end
  else
  begin
    MessageDlg ('Text not found',
      mtInformation, [mbOK], 0);
  end;
end;
```

Tipp

A fenti kód egyel növeli vagy csökkenti a pozíció értékeket, mert a Pos függvény egynek értelmezi a karakterlánc első karakterét, a SelStart tulajdonság pedig nullával kezdődik. Ha ugyanezt a kódot 16 bites Delphivel akarnánk elkészíteni, akkor a memo összes sorát végig kellene néznünk, mert a Pos függvény csak 255 karakteres Pascal karakterláncokra működött. A 32 bites Delphiben a memo teljes szövegét egyetlen hosszú Pascal karakterláncként értelmezhetjük.



13.12. ábra *Általános Find párbeszédablak, amit a CommDlg példában részben elkészítünk*

A Replace párbeszédablak keresés művelete annyira hasonló, hogy egy értékadás után akár az előző metódust is használhatjuk:

```
procedure TCommDlgForm.ReplaceDialog1Find(Sender:
                                                TObject);
begin
  FindDialog1.FindText := ReplaceDialog1.FindText;
  FindDialog1Find (ReplaceDialog1);
end;
```

A legjobb megoldás talán az, ha írnunk egy keresés metódust, amit meghívhatunk a Replace rutinhoz is. Ehelyett itt egy külön metódust írtunk a Replace gombhoz, amiben csak egyetlen új sor van:

```
if FoundPos > 0 then
  begin
    ...
    Memo1.SelText := ReplaceDialog1.ReplaceText;
  end
```

A többi párbeszédablakot legfeljebb néhány opció beállítása után futtathatjuk. A párbeszédablakokat még tovább alakíthatjuk, és ezt a példát is bővíthetjük azzal, hogy választhatunk normál és 3D megjelenítés között, megerősítést kérhetünk a felhasználótól a változtatásaira – ez különösen az állományokkal kapcsolatos párbeszédablakoknál kellhet –, és így tovább. Érdemes ezzel a témával bővebben foglalkozni, először a példa részleteiben elmélyedve, majd további saját beállítású általános párbeszédablakok hozzáadásával.

Új!

Még két komponens van a Delphi3 komponens palettájának Dialogs oldalán, az OpenPictureDialog, és a SavePictureDialog. Ezek az OpenDialog és SaveDialog komponensek változatai, amik grafikus állományok megnyitását és elmentését szolgálják azzal, hogy előzetest mutatnak az éppen kiválasztott képről.

Üzenetablakok sokasága

A Delphi üzenet és beviteli ablakok az előre definiált párbeszédablakok másik fajtáját képviselik. Egyszerű párbeszédablakok megjelenítésére Delphiben alapvetően hat eljárás és függvény szolgál:

- A MessageDlg függvény egy átalakítható üzenet ablakot jelenít meg, egy vagy több gombbal, és általában egy képpel. Ezt a függvényt az előző példákban viszonylag sűrűn használtuk.
- A MessageDlgPos függvény hasonlít a MessageDlg függvényre. A különbség, hogy az üzenet ablak a megadott pozíción jelenik meg, és nem a képernyő közepén.
- A ShowMessage eljárás még egyszerűbb üzenetablakot jelenít meg, aminek a címsorában az alkalmazás neve szerepel, és egyetlen OK gombja van. A ShowMessageFmt eljárás a ShowMessage egy változata, aminek ugyanaz a paraméterezése, mint a Format függvénynek. Ez megfelel annak, mintha a ShowMessage belsejében meghívtuk volna a Format-ot.
- A ShowMessagePos eljárás ugyanezt csinálja, de mi határozhatjuk meg az ablak pozícióját.

- Az InputBox függvény egy karakterláncot vár a felhasználótól. Itt az ablak címét, a kérdést, és az alapértelmezett választ adhatjuk meg.
- Az InputQuery függvény is egy karakterláncot vár. Az egyetlen eltérés az InputBox-hoz képest a szintaktika. Az InputQuery függvénynek logikai a visszatérési értéke attól függően, hogy a felhasználó OK-t vagy Cancel-t választott.

Az első két üzenet ablak összetettebb, több paramétere van, beleértve a gombok számát, az üzenetablak típusát, a súgókörnyezetet (ez akkor hasznos, ha van Súgó gomb), és esetleg a pozíciót is. A MessageDlg és a MessageDlgPos függvények a második, TMsgDlgType típusú, paramétertől függően többféle üzenetablakot jeleníthetnek meg:

```
type
  TMsgDlgType = (mtWarning, mtError, mtInformation,
    mtConfirmation, mtCustom);
```

Mindegyik típusú üzenetablaknak megvan a saját ikonja (sorban: sárga felkiáltó jel, vörös stop tábla, kék „i", zöld kérdőjel, és az utolsóhoz nincs kép). Ezeknek az üzenetablakoknak lehet egy vagy több gombja, ahogy azt a következő halmaz típus jelzi:

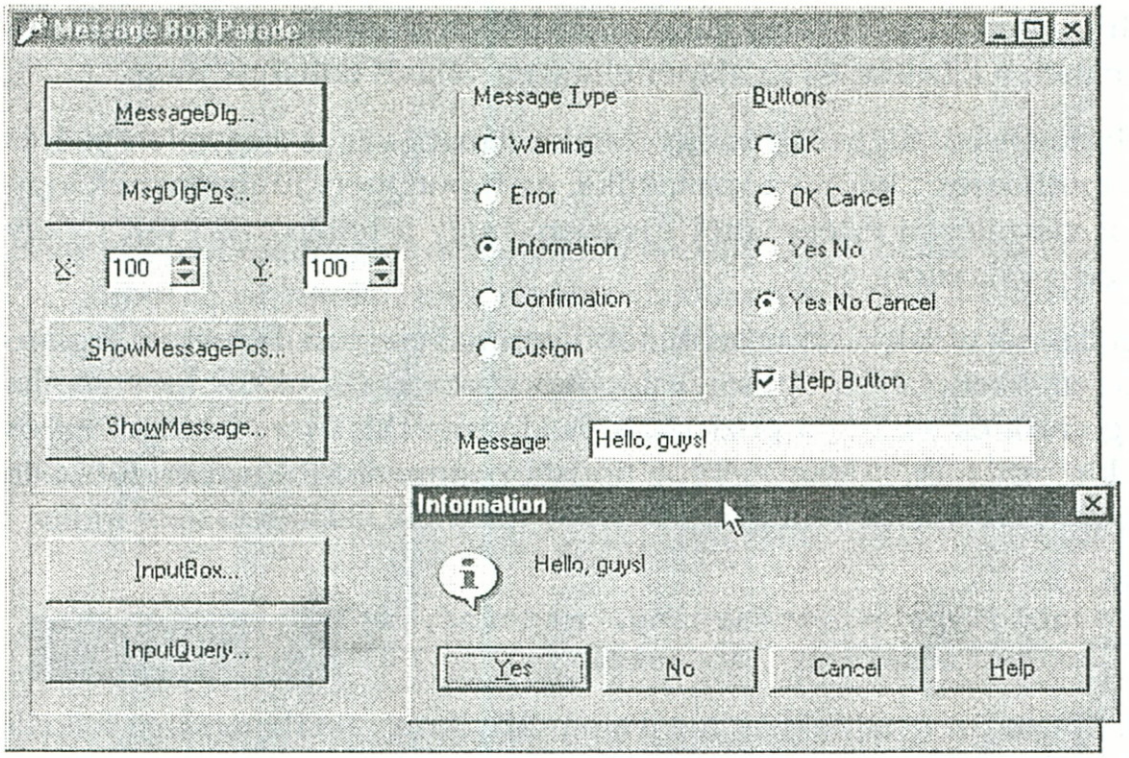
```
type

  TMsgDlgBtn = (mbYes, mbNo, mbOK, mbCancel,

    mbAbort, mbRetry, mbIgnore, mbAll, mbHelp);

  TMsgDlgButtons = set of TMsgDlgBtn;
```

A Delphi különböző üzenet ablakainak bemutatására létrehoztunk egy másik, a CommDlg példához hasonló programot. A MBParade példában sok beállítási lehetőségünk van – rádiógombok, jelölőnégyzetek, szerkesztőmezők, és beállítók – amikkel beállíthatjuk a gombok hatására megjelenő üzenetablakot. A 13.13. ábra alapján jobban elképzelhetjük a programot.

Egy MessageDlg ablaknál kiválaszthatjuk a stílusát, a gombjait – beleértve a súgó gombot is, és az üzenetet. A két helyre igazító függvény használja a két beállító értékét. Ami érdekes, az, hogy ha a „custom" (saját) típust választjuk, és csak OK gombot teszünk rá, akkor a MessageDlg ablak közönséges ShowMessage ablakká egyszerűsödik.

A program alapfelépítése a következő. A formnak két mezője tartalmazza a beállítások aktuális értékinek egy részét:

```
private
    MsgDlgType: TMsgDlgType;
    MsgButtons: TMsgDlgButtons;
```



13.13. ábra *Az MBParade példa főformja, példa üzenetablakkal*

Amikor a felhasználó kiválasztja az egyik rádió gombot, beállítjuk a megfelelő jelzőt, ahogy azt a két csoport rádiógombjaival összekapcsolt eljárások mutatják:

```
procedure TForm1.RadioWarningClick(Sender: TObject);
begin
  MsgDlgType := mtWarning;
end;

procedure TForm1.RadioOKCancelClick(Sender: TObject);
begin
  MsgButtons := mbOkCancel;
end;
```

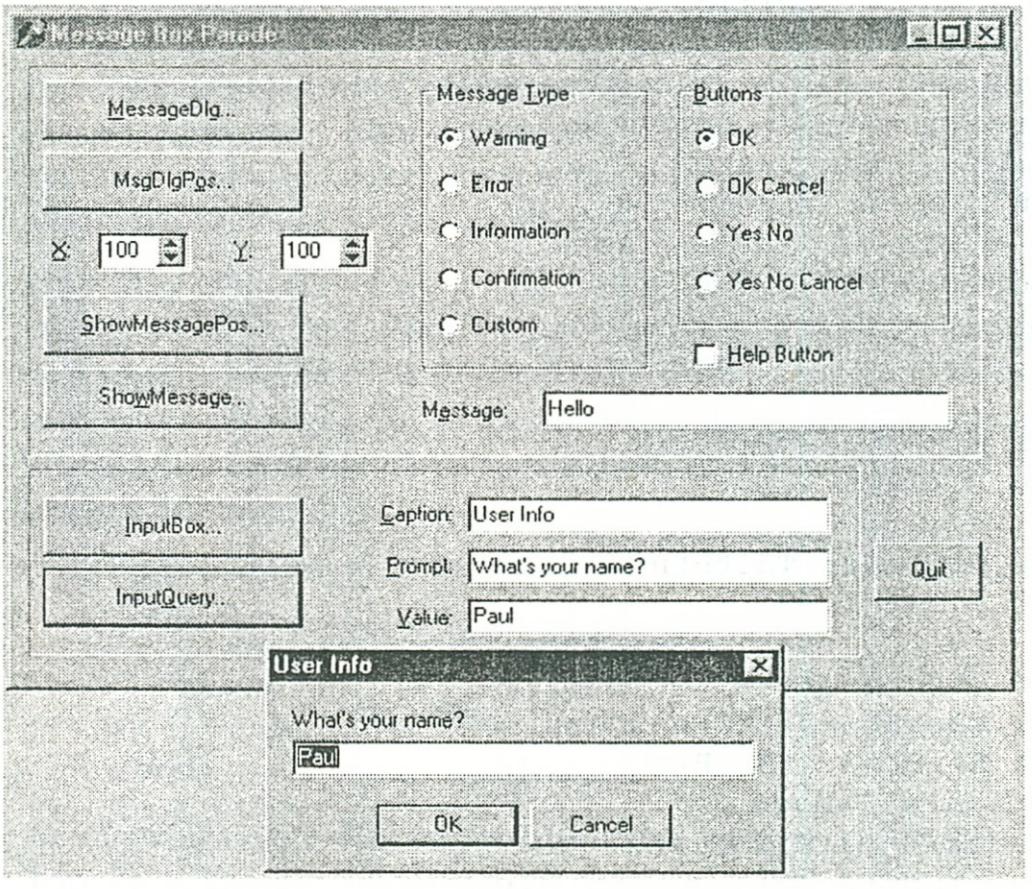
Ezeket a mezőket használjuk fel a párbeszédablakok megjelenítésére szolgáló gombok megnyomásakor. A következő metódus hívja a MessageDlgPos függvényt:

```
procedure TForm1.ButtonMsgDlgPosClick(Sender: TObject);
begin
  if CheckHelp.Checked then
    Include (MsgButtons, mbHelp);
  MessageDlgPos (Edit1.Text, MsgDlgType, MsgButtons,
    0, SpinX.Value, SpinY.Value);
end;
```

A beviteli ablakok csak a form második felében lévő három szerkesztőmező értékét használják. Attól eltekintve, hogy a programjuk eltérő, a megjelenésük egyforma, ahogy az a 13.14. ábrán is látható. Íme a két hívás:

```
procedure TForm1.ButtonInputBoxClick(Sender: TObject);
begin
  EditValue.Text := InputBox (EditCaption.Text,
    EditPrompt.Text, EditValue.Text);
end;

procedure TForm1.ButtonInputQueryClick(Sender: TObject);
var
  Text: String;
begin
  Text := EditValue.Text;
  if InputQuery (EditCaption.Text,
      EditPrompt.Text, Text) then
    EditValue.Text := Text;
end;
```

Ezután a forrás többi részét már könnyű megérteni, ha átnézzük, de ennél fontosabb, hogy futtassuk, és ezáltal megismerhessük a különböző függvények paramétereinek hatását.



13.14. ábra *A MBParade példa főformja nyitott InputQuery üzenetablakkal*

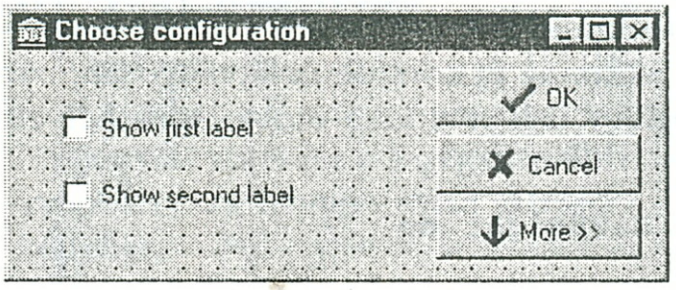
Bővíthető párbeszédablakok

Néhány párbeszédablak különböző komponenseket jelenít meg a felhasználónak. Ezeket logikai oldalakra oszthatjuk, amit a Delphi a TabControl (füles kontrol) és a Notebook (jegyzettömb) komponensekkel támogat, erről a tizennegyedik fejezetben lesz szó. Más esetekben a párbeszédablak egy részét el lehet rejteni, hogy segítsük a kezdő felhasználókat. Arra, hogy az összetettebb információkat csak akkor mutassuk meg, ha a felhasználó kéri, alapvetően kétféle megközelítés létezik: készíthetünk másodlagos párbeszédablakot, amit akkor jelenítünk meg, ha a felhasználó az Advanced... gombra kattint, vagy megnövelhetjük a párbeszédablak méretét, hogy több kontroll férhessen el rajta, amikor a felhasználó a More gombot nyomja meg.

A második megközelítés kétségkívül érdekesebb, de több figyelmet is igényel. Ilyen módon fogjuk kiterjeszteni a párbeszédablakokkal foglalkozó példát, a Dialog1-et, és készítjük el a More programot. Először bővítsük ki a párbeszédablakot új kontrollokkal: egy, a 13.15. ábrán is látható, More gombbal, és egy teljesen új form résszel, amin két új jelölőnégyzet áll, italic (dőlt) és bold (félkövér) címkékkel, ezekkel lehet megváltoztatni a főform szövegstílusát. Miután hozzáadtuk az új kontrollokat, át kell méreteznünk a párbeszédablakot, hogy az új összetevők kikerüljenek a látható részről:

```
object ConfigureDialog: TConfigureDialog
  BorderStyle = bsDialog
  Caption = 'Choose configuration'
  ClientHeight = 106 // notice this value !!!
  OnActivate = FormActivate
  OnCreate = FormCreate
  object Label1: TLabel
    Top = 112 // out of range !!!
    AutoSize = False
    Caption = 'Here comes the rest of the dialog
                    box, with some new controls...'
  end
  object BitBtn1: TBitBtn
    Kind = bkOK
  end
  object BitBtn2: TBitBtn
    Kind = bkCancel
  end
  object CheckBox1: TCheckBox...
  object CheckBox2: TCheckBox...
  object BitBtn3: TBitBtn
    Caption = '&More >>'
    OnClick = BitBtn3Click
  end
```

```
    object ItalicCheckBox: TCheckBox
      Top = 152 // out of range !!!
      Caption = '&Italic'
      Enabled = False
    end
    object BoldCheckBox: TCheckBox
      Top = 152  // out of range !!!
      Caption = '&Bold'
      Enabled = False
    end
  end
```



13.15. ábra *A More példa párbeszédablaka tervezés közben. Néhány komponens nem látszik, mert a kereten kívülre kerültek*

A nem látható részen lévő komponenseket – amik mellett megjegyzés áll – célszerű kikapcsolni, mert különben a felhasználó a Tab billentyűvel rájuk válthat akkor is, amikor nem látszanak. Másik megoldásként kikapcsolhatjuk a TabStop tulajdonságukat. Ezt a tulajdonságot – Enabled, vagy TabStop – a form megnagyobbításakor újra igazra állítjuk.

Most készítsük el a More gomb megnyomásakor a párbeszédablak nagyítását végző, valamint a két jelölőnégyzetet kezelő programot. Az átméretezéshez szükségünk lesz néhány új mezőre a formon – ezek az OldHeight (régi magasság) és NewHeight (új magasság) mezők –, amikben eltárolhatjuk a form kliensterületének két lehetséges magasságát. Ezeket az értékeket a form készítésekor állíthatjuk be:

```
procedure TConfigureDialog.FormCreate(Sender: TObject);
begin
  OldHeight := ClientHeight;
  NewHeight := ItalicCheckBox.Top +
    ItalicCheckBox.Height + BitBtn1.Top;
end;
```

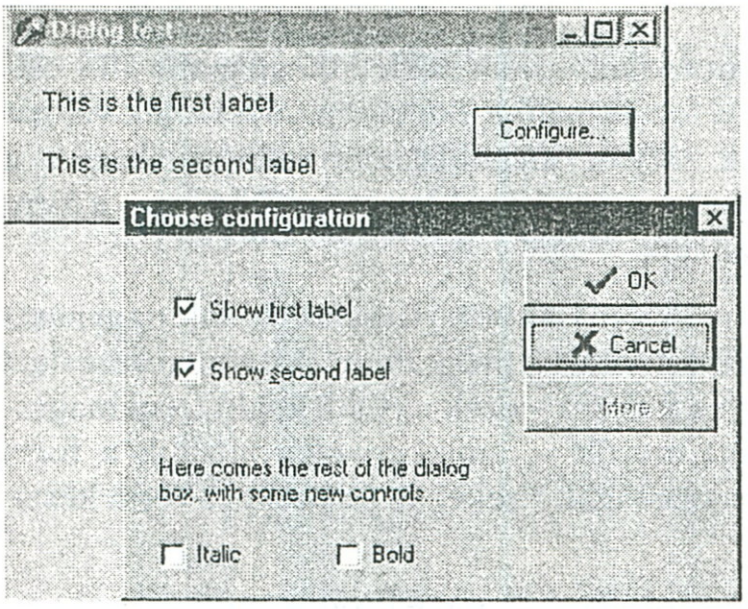
Az új magasságot úgy határoztuk meg, hogy a jelölőnégyzet aljához (amit a Top és a Height tulajdonságok összeadásával kapunk meg) hozzáadjuk a

legfelső képes gomb feletti margót. A valódi átméretezés akkor történik, ha a felhasználó megnyomja a More gombot. Íme az első változata:

```
procedure TConfigureDialog.BitBtn3Click(Sender:TObject);
begin
  BitBtn3.Enabled := False;
  BoldCheckBox.Enabled := True;
  ItalicCheckBox.Enabled := True;
  ClientHeight := NewHeight;
end;
```

A végeredményt a 13.16. ábrán láthatjuk. Ha látványosabb hatást akarunk, akkor megtehetjük, hogy a magasságot egyesével növeljük, és nem rögtön a végleges értékre állítjuk. Ha erre ciklust írunk, ami megnöveli a kliens magasságot, és mindig újra festi az ablakot, akkor az új kontrollok látványosan, bár lassabban jelennek meg. A fenti BitBtn3Click metódus utolsó sora helyett ilyenkor ez szerepel:

```
for I := ClientHeight to NewHeight do
begin
  ClientHeight := I;
  Update;
end;
```



13.16. ábra *A More példa párbeszédablaka átméretezés után*

Amikor a párbeszédablakot aktiváljuk (OnFormActivate esemény), a magasságát visszaállítjuk, és kikapcsoljuk a rejtett komponenseket, hogy ne lehessen TAB-bal rájuk menni, és bekapcsoljuk a More gombot:

```
procedure TConfigureDialog.FormActivate(Sender:TObject);
begin
  ClientHeight := OldHeight;
  BitBtn3.Enabled := True;
  BoldCheckBox.Enabled := False;
  ItalicCheckBox.Enabled := False;
end;
```

Erre a kódra mindig szükség van a párbeszédablak megjelenítésekor, hogy kezdetben a kis változat látszódjon.

Névjegy készítése

A Windows alkalmazásoknak általában van Névjegyük, ahol olyan információkat találhatunk, mint a program verzió száma, a szerző neve, és így tovább. Névjegy készítésére a legegyszerűbb megoldás a MessageDlg függvény. Ezzel azonban csak nagyon rövid szöveget írhatunk ki, és nem használhatunk saját képeket.

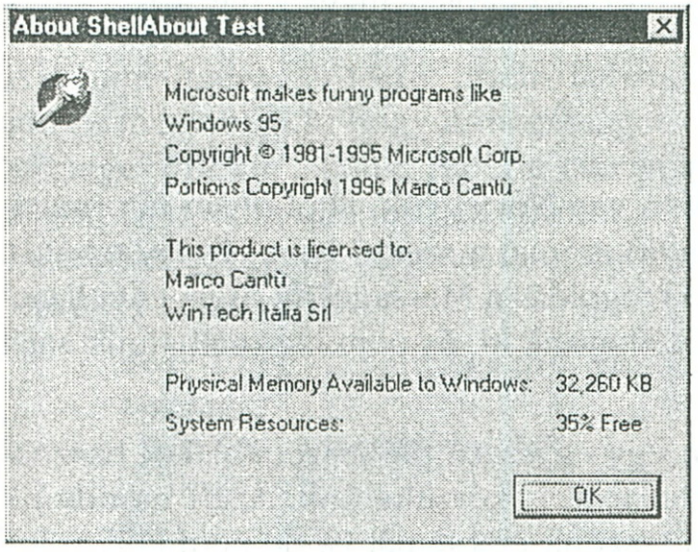
Ezért Névjegy készítésekor általában egy egyszerű párbeszédablakot használunk, például amilyet a Delphi sablonjaival is készíthetünk. Azért mondtunk egyszerűt, mert lehet, hogy a formra teszünk emblémát is, de nem kell mögé sok kód. A legtöbb, amire szükség lehet, hogy megjelenítsük a rendszer-információkat, mint a Windows verziója, a szabad memória mérete, vagy valami más információ, mint a regisztrált felhasználó neve.

A rendszer Névjegyének használata

Rendhagyó alternatívaként választhatjuk a számos Windows alkalmazás általános párbeszédablakát. Ha az Intéző, vagy bármely más, a Windowsba beépített alkalmazás Névjegyét megnézzük, akkor ugyanazt a párbeszédablakot láthatjuk. Ez a rendszerablak, bár nem dokumentáltan, megvolt már a Windows 3.1-ben is. A Win32 API-ban és a Delphi 32 bites változatában már némileg dokumentált a párbeszédablak használata: A ShellApi egység ShellAbout függvényével megjeleníthetjük ezt az ablakot, és némiképp át is alakíthatjuk. Ezt a függvényt hívjuk meg a ShAbout példában is, amikor a felhasználó megnyomja a form egyetlen gombját:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShellAbout (Handle,
    'About ShellAbout Test#makes funny programs like',
    'Portions Copyright 1996 Marco Cantù',
    Application.Icon.Handle);
end;
```

Ami a Windows API súgó állományban nem szerepel, az az, hogy az első string paraméter két, „#" jellel elválasztott részből áll. A paraméter első része a rendszer Névjegy ablak címsora lesz, míg a második rész pedig a szöveg rögzített részei között jelenik meg. A hatás nagyon humoros, ahogy azt a 13.17. ábrán is láthatjuk. Valójában ez nem túl hasznos, mert a párbeszédablakunk összhatását rontja a Microsoft neve, amit a valódi alkalmazásokban nem szeretnénk.

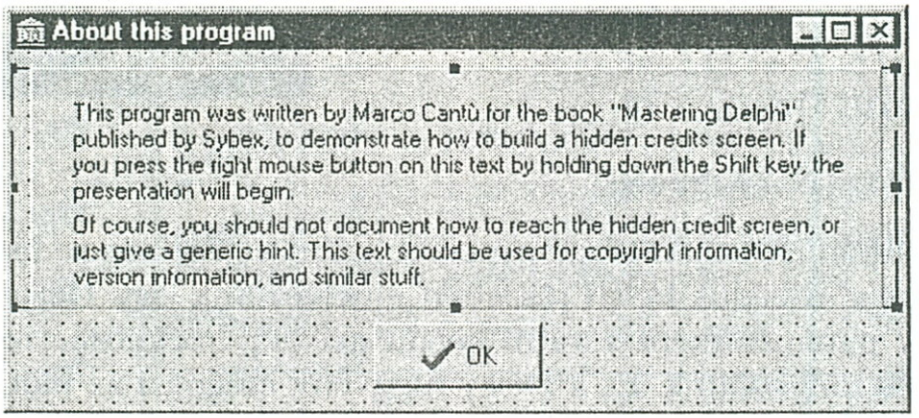


13.17. ábra *Az ShAbout program Névjegye*

Saját rejtett képernyő készítése

Úgy tűnik, hogy elkészültünk az első Névjegyünkkel. Ezután hozzáadhatunk saját rejtett képernyőt, amit a Delphiben, és sok más alkalmazásban is láthatunk. Rejtett képernyő hozzáadásának több oka lehet. Ha egy nagy cégnek dolgozunk, akkor ezzel bebizonyíthatjuk, hogy részt vettünk abban a projektben, és ha ez a projekt sikeres volt, akkor ez segíthet később, ha munkát keresünk. Máskor a rejtett Névjegyeket érdemes megnézni, mert remek alkalmat adnak arra, hogy a vetélytársainkat megtréfáljuk. Komolyabb indok lehet, hogy a rejtett „szerzők felsorolása" ablak bebizonyíthatja, hogy ki készítette a programot.

Készítettünk egy egyszerű programot, amin megláthatjuk, hogy hogyan hozhatunk létre rejtett képernyőket. Ahogy az a 13.18. ábrán is látható, a párbeszédablak egy két címkét tartalmazó panelből áll. A panelen akárhány komponens lehet, akár szövegesek és grafikusak is, sőt a szövegek egy részét akár futás időben is előállíthatjuk. Az egyetlen különlegesség, amire a rejtett rész megjelenítésekor szükségünk van, egy PaintBox komponens, ami a form egy részét takarja.



13.18. ábra *A Credits példa Névjegye tervezés alatt*

Amikor a felhasználó egy meghatározott összetett feladatot elvégez (jelen esetben a SHIFT lenyomása mellett jobb gombbal kattint a felső címkén), a panelt elrejtjük, és valami megjelenik a képernyőn. A legegyszerűbb megoldás, ha szöveget rajzolunk a form felületére, vagyis a „canvas"-ra:

```
procedure TAboutBox.Label1MouseDown(Sender: TObject;
  Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if (Button = mbRight) and (ssShift in Shift) then
  begin
    Panel1.Visible := False;
    PaintBox1.Canvas.Font.Name := 'Arial';
    PaintBox1.Canvas.Font.Size := 20;
    PaintBox1.Canvas.TextOut (40, 50, 'Author: Marco
      Cantù');
    PaintBox1.Canvas.TextOut (40, 100, 'Version 1.0');
  end;
end;
```

Látványosabb rejtett képernyőt kapunk, ha szöveget görgetünk, ahogy azt a Credits példa végleges változatában tesszük. Nézzük meg, hogy a sorok helyzete a szöveg magasságától függ, amit a PaintBox komponens Canvas-ának TextHeight (szöveg magasság) metódusával kapunk meg:

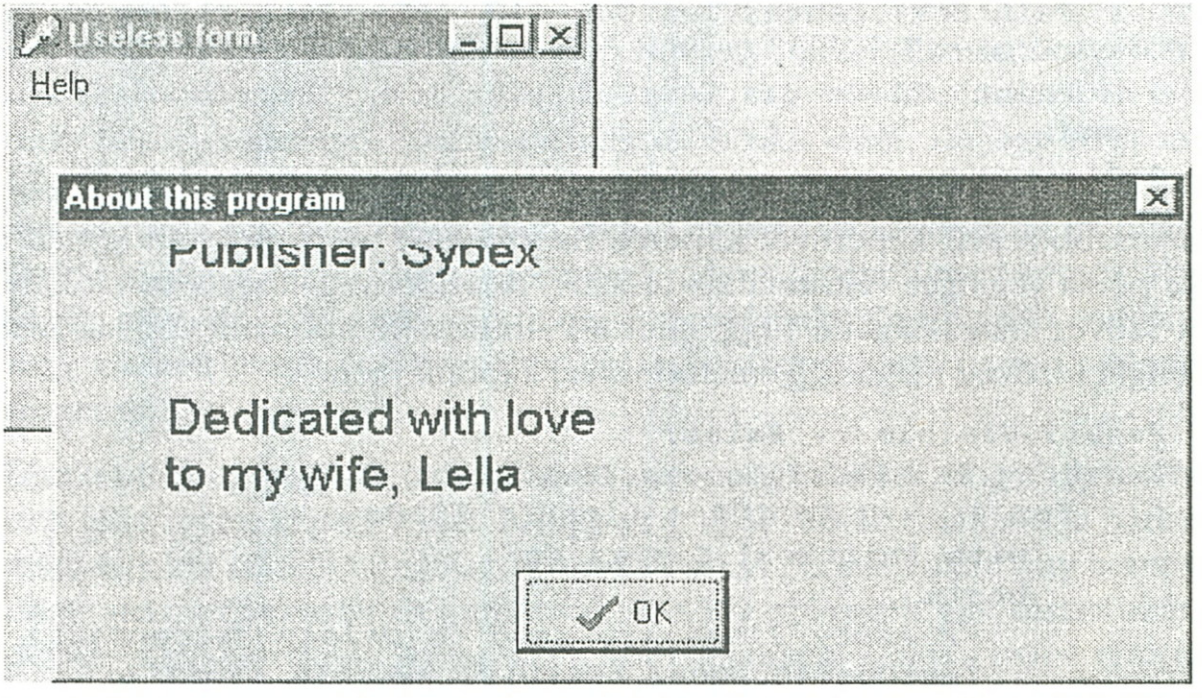
```
Panel1.Visible := False;
LineH := PaintBox1.Canvas.TextHeight ('0');
    for I := 0 to 100 + LineH * 10 do
      with PaintBox1.Canvas do
      begin
        // empty lines are used to delete descendants
        TextOut(40,100-I,'CREDITS example from:');
        TextOut(40,100+LineH-I,'"Mastering Delphi"');
        TextOut(40,100+LineH*2-I,'                  ');
```

```
        ...
        // wait 5 milliseconds
        Delay(0.5);
      end;
    Panel1.Visible := True;
```

A kód hatását a 13.19. ábrán láthatjuk. Hogy gyorsabb számítógépeken se legyen túl gyors a görgetés, a ciklus belső részében elhelyeztünk egy Delay függvényhívást, aminek paraméterként a várni kívánt másodperceket illetve ezredmásodperceket kell megadnunk. Ez a Delay függvény megnézi az időt, és utána addig vár, ameddig a megadott másodpercek, illetve századmásodpercek el nem teltek:

```
procedure Delay (Seconds, MilliSec: Word);
var
  TimeOut: TDateTime;
begin
  TimeOut := Now + EncodeTime (0,
    Seconds div 60, Seconds mod 60, MilliSec);
  // wait until the TimeOut time
  while Now < TimeOut do
    Application.ProcessMessages;
end;
```

A ciklusban meghívjuk a globális Application objektum ProcessMessages (üzenetfeldolgozás) metódusát, hogy a Windows elküldhesse, és feldolgozhassa a szükséges újrarajzoló üzeneteket. Ez a váró eljárás nagyon általános, ezért máshol is könnyen felhasználhatjuk.



13.19. ábra *A névjegy rejtett készítők képernyője, a Credits példában*

Tipp

Vegyük észre az előző példa egy másik vonatkozását is. Olyan kódot írtunk, ami a párbeszédablak felületére rajzolt. Bár ez nem igazán jellemző, a párbeszédablakok tartalmazhatnak grafikus információkat is, reagálhatnak egér eseményekre, mint bármelyik form, hiszen a párbeszédablak is form.

Most, hogy ismerjük ezt is, vigyázzunk, hogy a csodálatos rejtett képernyők elkészítése nem menjen a hasznos kód rovására, mert igaz ugyan, hogy a Delphivel időt takaríthatunk meg, de azt az időt jobban is felhasználhatjuk.

Nyitóképernyő készítése

Sok alkalmazás rendelkezik olyan nyitóképernyővel, ami a főform megjelenése előtt látható, ettől az alkalmazás egyrészt gyorsabbal tűnik, mert a felhasználó töltés közben is lát valamit, másrészt jól is néz ki. Néha ugyanezt az ablakot használják az alkalmazás Névjegyeként.

Hogy bemutassunk egy példát, ahol a nyitóképernyő különösen hasznos, készítettünk egy programot, ami egy prímszámokkal teli listadobozt jelenít meg. Ezeket a prímszámokat a program induláskor számítja ki, ezért akkor jelennek meg, amikor a form megjelenik:

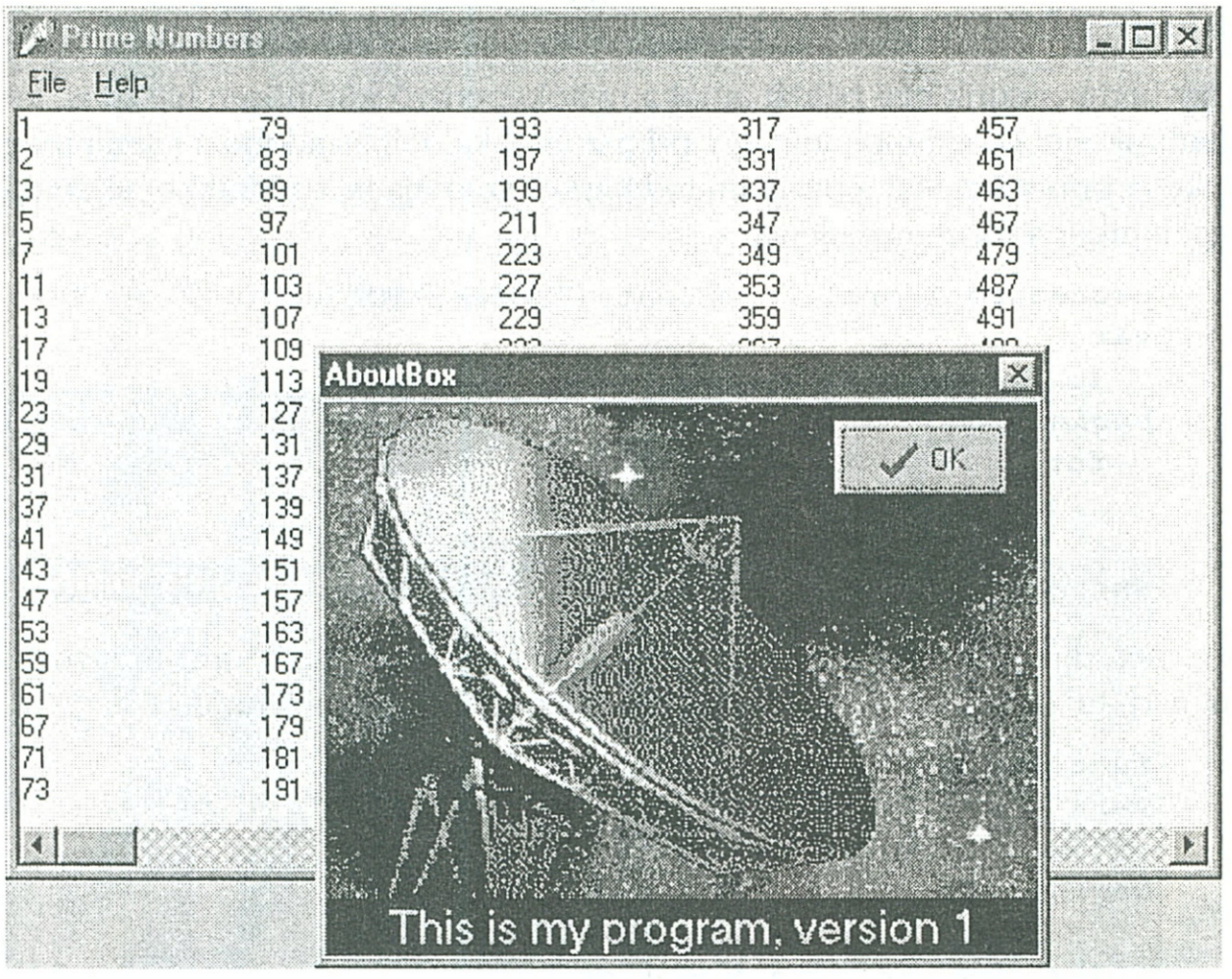
```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 10000 do
    if IsPrime (I) then
      ListBox1.Items.Add (IntToStr (I));
end;
```

Ez a metódus meghívja a szintén általunk készített IsPrime (Prím) függvényt, ami nem túl hatékony, de most a céljainkhoz lassúnak kell lennie:

```
function IsPrime (N: Integer): Boolean;
var
  Test: Integer;
begin
  IsPrime := True;
  for Test := 2 to N - 1 do
    if (N mod Test) = 0 then
    begin
      IsPrime := False; break;
    end;
end;
```

A számokat egy, a teljes kliens területet eltakaró, többoszlopos listadobozhoz adjuk, amit a 13.20. ábrán is láthatunk. Íme a form szöveges leírásának rövid összegzése:

```
object Form1: TForm1
  Caption = 'Prime Numbers'
  Menu = MainMenu1
  OnCreate = FormCreate
  object ListBox1: TListBox
    Align = alClient
    Columns = 5
  end
  object MainMenu1: TMainMenu
    object File1: TMenuItem...
      object Exit1: TMenuItem...
    object Help1: TMenuItem...
      object About1: TMenuItem...
  end
end
```



13.20. ábra *A Splash példa főformja a menüből előhívott Névjeggyel*

A Splash0 futtatásakor láthatjuk a program legnagyobb hibáját, mégpedig azt, hogy a FormCreate metódusban elhelyezett kezdeti műveletek hosszú ideig tartanak. Amikor elindítjuk a programot, a főablak megjelenéséig másodperceket kell várnunk egy átlagos Pentiumon. Ha a gépünk túl gyors, vagy túl lassú, akkor a FormCreate metódusban lévő ciklus felső határának módosításával a programot gyorsíthatjuk vagy lassíthatjuk.

A program egy egyszerű párbeszédablakot tartalmaz, egy képpel, egy felirattal, és egy képes gombbal, amit egy, az ablak teljes felszínét elfoglaló panelra tettünk. Ez a form jelenik meg a Help menü About pontjánál.

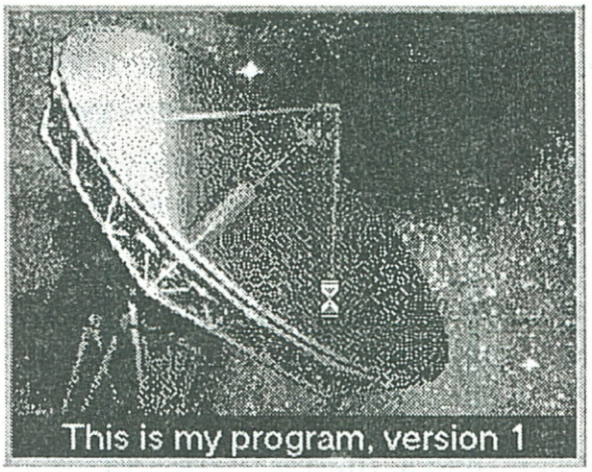
A Névjegy ablak OnClick eseménykezelője is érdekes, mert a párbeszédablak nem a program indulásakor keletkezik – ugyanis kikapcsoltuk a Project Options párbeszédablak **Forms** oldalán –, hanem az első megjelenése előtt, és a megjelenítés után a kód nem semmisíti meg, mert később még szükség lehet rá. Felmerülhet a kérdés, hogy honnan állapítjuk meg, hogy létre kell-e hoznunk a formot. Ehhez meghívjuk az Assigned metódust, ami leellenőrzi, hogy az objektum „nil"-e:

```
procedure TForm1.About1Click(Sender: TObject);
begin
  if not Assigned (AboutBox) then
    AboutBox := TAboutBox.Create (Application);
  AboutBox.ShowModal;
end;
```

Ez az általánosan használt technika arra, hogy a formot csak akkor hozzuk létre, ha tényleg szükség van rá, anélkül, hogy minden használat után megsemmisítenénk. Mi azonban azt szeretnénk, hogy a Névjegy a program indulásakor megjelenjen; ezt a hatást láthatjuk a Splash1 vagy a Splash2 példákban, amik különböző módszerekkel elkészített nyitóképernyőket használnak.

Elsőként egy új metódust adunk a TAboutBox osztályhoz. Ez a MakeSplash nevű metódus, a form néhány tulajdonságát változtatja meg, hogy alkalmassá tegye nyitóképernyőnek. Eltávolítja a keretet és a címsort, elrejti az OK gombot, a panel keretét megvastagítja, hogy ezzel helyettesítse a form keretét, majd megjeleníti a formot, és azonnal újra is festi – ennek hatása a 13.21. ábrán látható:

```
procedure TAboutBox.MakeSplash;
begin
  BorderStyle := bsNone;
  BitBtn1.Visible := False;
  Panel1.BorderWidth := 3;
  Show;
  Update;
end;
```



13.21. ábra *A Splash1 példa nyitóképernyője némiképp eltér az eredeti Névjegytől, amit a 13.20. ábrán láthattunk*

Ezt a metódust azután hívjuk meg, hogy a Splash1 példa projekt állományában létrejött a form. A kód a többi form – ebben a példában csak a főform – létrehozása előtt fut le, és a nyitóképernyőt el is távolítjuk az alkalmazás elindítása előtt. Ezek a műveletek egy `try-finally` blokkban kapnak helyet. Itt következik a Splash2 példa teljes projektállománya:

```
program Splash;

uses
  Forms,
  MainSpF in 'MainSpF.pas' {Form1},
  AboutF in 'AboutF.pas' {AboutBox};

{$R *.RES}

var
  SplashAbout: TAboutBox;

begin
  Application.Initialize;
  // create and show the splash form
  SplashAbout := TAboutBox.Create (Application);
  try
    SplashAbout.MakeSplash;
    // standard code...
    Application.CreateForm(TForm1, Form1);
    // get rid of the splash form
    SplashAbout.Close;
  finally
    SplashAbout.Free;
  end;
  Application.Run;
end.
```

Ennek a megközelítésnek azonban csak akkor van értelme, ha a kezdő kód lassú, és az alkalmazás főformjának elkészítése hosszú időt vesz igénybe, akár, mint ebben az esetben is, az inicializáló kód hossza, vagy egy adatbázis kapcsolat létrehozása miatt.

Másik megközelítés, hogy a nyitóképernyőt hosszabb ideig mutatjuk, és egy időzítő segítségével tüntetjük el. Ezt a technikát használtuk a Splash2 példában. Ez a példa más megközelítést használ a nyitóképernyő létrehozására: A projekt állomány helyett a főform FormCreate metódusának elején készítjük el:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
  SplashAbout: TAboutBox;
begin
  // create and show the splash form
  SplashAbout := TAboutBox.Create (Application);
  SplashAbout.MakeSplash;
  // standard code...
  for I := 1 to 10000 do
    if IsPrime (I) then
      ListBox1.Items.Add (IntToStr (I));
  // get rid of the splash form, after a while
  SplashAbout.Timer1.Enabled := True;
end;
```

Az időzítőt a metódus befejezése előtt kapcsoljuk be. Amikor a kiválasztott intervallum letelt – a példában három másodperc – az OnTimer esemény meghívódik, amit a Névjegy form kezel le, úgy, hogy becsukja, és megsemmisíti önmagát:

```
procedure TAboutBox.Timer1Timer(Sender: TObject);
begin
  Close;
  Free;
end;
```

Már csak egy dolgot kell kijavítanunk, a főform ugyanis később jelenik meg, és a nyitókép előtt, hacsak nem tesszük azt a legfelsővé. Emiatt a Splash2 példa MakeSplash metódusába a következő sort írjuk:

```
FormStyle := fsStayOnTop;
```

Vizuális form öröklés

A Delphi 2 egyik legfontosabb újítása a vizuális form öröklés volt. Ezt egyszerűen összefoglalhatjuk úgy, hogy örököltethetünk egy formot egy már meglevőből, hozzáadhatunk új komponenseket, vagy már létezők tulajdonságait megváltoztathatjuk. De mi a vizuális form öröklés valódi előnye?

Ez teljes mértékben az éppen fejlesztett alkalmazástól függ. Ha több formunk van, amik közül több nagyon hasonló, vagy csak vannak közös elemeik, akkor a közös elemeket, és eseménykezelőket egy alapformra helyezhetjük, és a különleges összetevőket és szolgáltatásokat csak az alosztályokban készítjük el. Például, készíthetünk egy szabványos formot, eszköztárral, emblémával, alapvető kóddal átméretezésre, és bezárásra, valamint néhány Windows üzenetkezelővel, és ezt egy alkalmazás összes formjának őseként felhasználhatjuk.

A vizuális form öröklést felhasználhatjuk még arra, hogy az alkalmazásunk felületét a különböző ügyfelek számára átalakítsuk, anélkül, hogy akár a forrást, akár a formot definiáló kódot duplikálnánk, hiszen elég, ha az alapértelmezett formból örököltetünk egy ügyfél specifikus változatot. Ugyanígy készíthetjük el a programunk különböző nyelvű változatait a komponensek feliratainak lefordításával. Ennek megfelelően akár adatelérést szolgáló komponenseket is tehetünk az alap formra, de gyakran érdemesebb egy másodlagos formot, vagy adatmodult (Data Module) használni, mint ezeket is örököltetni.

A vizuális örököltetés legnagyobb előnye, hogy később megváltoztathatjuk az eredeti formot, és a leszármazott formok automatikusan követik a változásokat. Ez az objektum-orientált programnyelvek öröklésének egyik legismertebb előnye. Emellett azonban van egy másik hatás is, a polimorfizmus, vagyis, hogy az alap formhoz virtuális metódusokat adhatunk, amiket a leszármazott formokban felüldefiniálunk, és ezután mindegyiken hivatkozhatunk a metódusok bármelyikére.

Figyelem!

Bár a vizuális form öröklés nagyon hatékony technika, mégis megvannak a buktatói. Néhány tanács azonban segíthet a problémák elkerülésében: először, soha ne változtassuk meg a komponens nevét, se az alap, se a leszármazott formokon; másodszor, mindig legyen nyitva az éppen szerkesztett form őse is. A problémák általában az alap és a leszármazott form közötti szinkronizáció elvesztéséből adódnak.

Öröklés alapformból

A vizuális form örököltetés szabályai nagyon egyszerűek, ha az örököltetéssel tisztában vagyunk. Alapjában a leszármazott formnak ugyanazok a komponensei vannak, mint a szülő formnak, esetleg néhány új is, és az örökölt komponensek tulajdonságait megváltoztathatjuk. Ha a leszármazott form valamely komponensének tulajdonságát megváltoztatjuk, akkor a szülő formon végzett módosításoknak nem lesz hatása, lehetőség van azonban a két érték újraszinkronizálására az Object Inspector **Revert to Inherited** (visszaállás az eredetileg örököltre) helyi menü parancsával. Ugyan ez érhető el akkor is, ha a két tulajdonságot azonos értékre állítjuk, és újrafordítjuk a kódot. Esetleges megoldásként használhatjuk azt, hogy a leszármazott form szöveges leírásából eltávolítjuk a tulajdonság megváltoztatását végző sort (hamarosan meglathatjuk az állomány felépítését). A komponensek mellett az új form az alapform minden metódusát is örökli, beleértve az eseménykezelőket is. A leszármazott formhoz adhatunk új kezelőket, vagy felülbírálhatjuk a már létezőket.

Hogy bemutassuk a vizuális form örököltetést, készítünk egy nagyon egyszerű, VFI-nek nevezett alkalmazást. Először új projektet nyitunk, és a főformjára négy gombot teszünk. Ezután kiválasztjuk a **File** menü **New** pontját, és a New Items párbeszédablakban azt az oldalt nyitjuk meg, amit a projekt neve jelez (ez látható a 13.22. ábrán). Itt választhatjuk ki az örököltetni kívánt formot. Az új formon ugyanaz a négy gomb van. Íme az új form kezdeti szöveges leírása:

```
inherited Form2: TForm2
  Caption = 'Form2'
end
```

Itt pedig a kezdeti osztálydeklaráció, amiben láthatjuk, hogy az alaposztály nem a már megszokott Tform, hanem a jelenlegi kiinduló formunk:

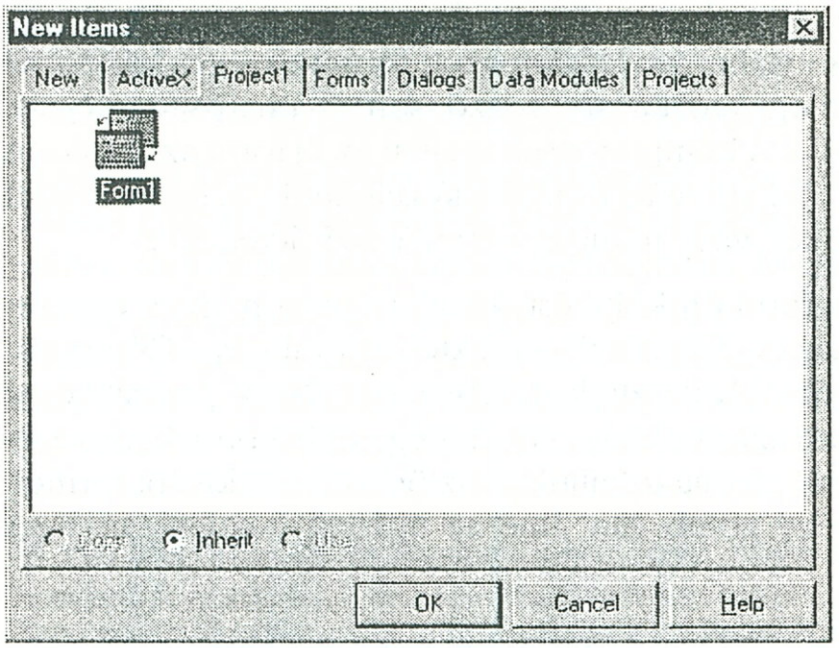
```
type
  TForm2 = class(TForm1)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Vegyük észre az inherited (örökölt) kulcsszót, és a tényt, hogy a formnak vannak komponensei, bár ezeket az ős formon definiáltuk. Ha elmozgatjuk a formot, és megváltoztatjuk az egyik gomb feliratát, akkor a szöveges leírás is ennek megfelelően változik:

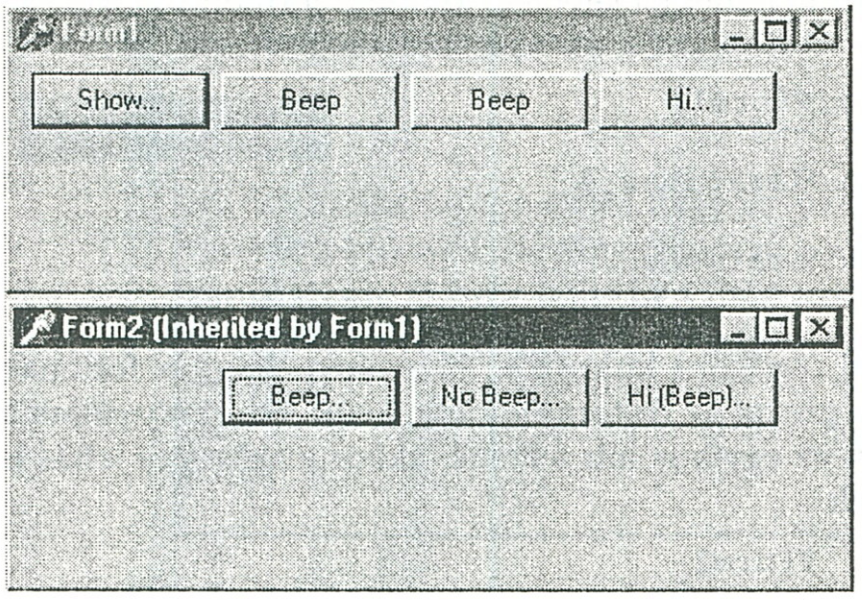
```
inherited Form2: TForm2
  Left = 313
  Top = 202
```

```
    Caption = 'Form2'
    inherited Button2: TButton
      Caption = 'Beep...'
    end
  end
```

Itt csak a megváltozott értékek szerepelnek, ezért tudjuk az ősbeli értékekre visszaváltoztatni ezeket azzal, ha kitöröljük a változásokat innen, ahogy előbb említettük. Amint az a 13.23. ábrán is látszik, a legtöbb gomb feliratát megváltoztattuk.



13.22. ábra *A New Items párbeszédablak segítségével készíthetünk leszármazott formokat*



13.23. ábra *A VFI példa két formja futás közben*

Az első form mindegyik gombjának van egy egyszerű OnClick eseménykezelője. Az első gomb a leszármazott formot mutatja meg a Show metódus meghívásával, a második és a harmadik a Beep eljárást hívja meg, az utolsó gomb egy egyszerű üzenetet jelenít meg a ShowMessage('Hi') függvényhívással.

Mi történik a leszármazott formon? Az első teendő az első gomb eltávolítása lenne, mert a másodlagos form már látszik, de leszármazott formról nem törölhetünk komponenseket. Erre lehet megoldás, ha a komponenst a helyén hagyjuk, de láthatatlanná tesszük, ahogy azt a 13.24. ábráról már kitalálhattuk. A másik három gomb látható, de más eseménykezelőjük van. Ezt egyszerűen megtehetjük, ugyanis ha a leszármazott form valamely gombjának OnClick eseményét kiválasztjuk – például azzal, hogy duplán kattintunk rajta –, egy üres metódust kapunk, ami némiképp eltér az alapértelmezettől:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  inherited;
end;
```

Az `inherited` (örökölt) kulcsszó az ős form megfelelő eseménykezelőjének meghívását helyettesíti. Ezt a kulcsszót a Delphi mindig hozzáadja az eseménykezelőhöz, akkor is, ha ezt az ős osztályban nem definiáltuk, ami érthető is, hiszen később még definiálhatjuk, ám akkor is, ha a komponens nem volt meg az ős osztályban, aminek, látszólag, nincs sok értelme.

Nagyon egyszerű az alapform kódját lefuttatni, más utasításokkal kibővítve:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  inherited;
  ShowMessage ('Hi');
end;
```

De nem ez az egyetlen megoldás. Másik megközelítésként írhatunk teljesen új eseménykezelőt, úgy, hogy az ős osztály kódját nem futtatjuk, ahogy azt a VFI példa harmadik gombjánál tettük:

```
procedure TForm2.Button3Click(Sender: TObject);
begin
  ShowMessage ('Hi');
end;
```

Ismét másik lehetőség, hogy az alaposztály metódusát néhány előkészítő utasítás után hívjuk meg, esetleg csak bizonyos feltételek esetén, vagy meghívhatjuk az alaposztály egy másik eseménykezelőjét, ahogy azt a negyedik gomb esetében tettük:

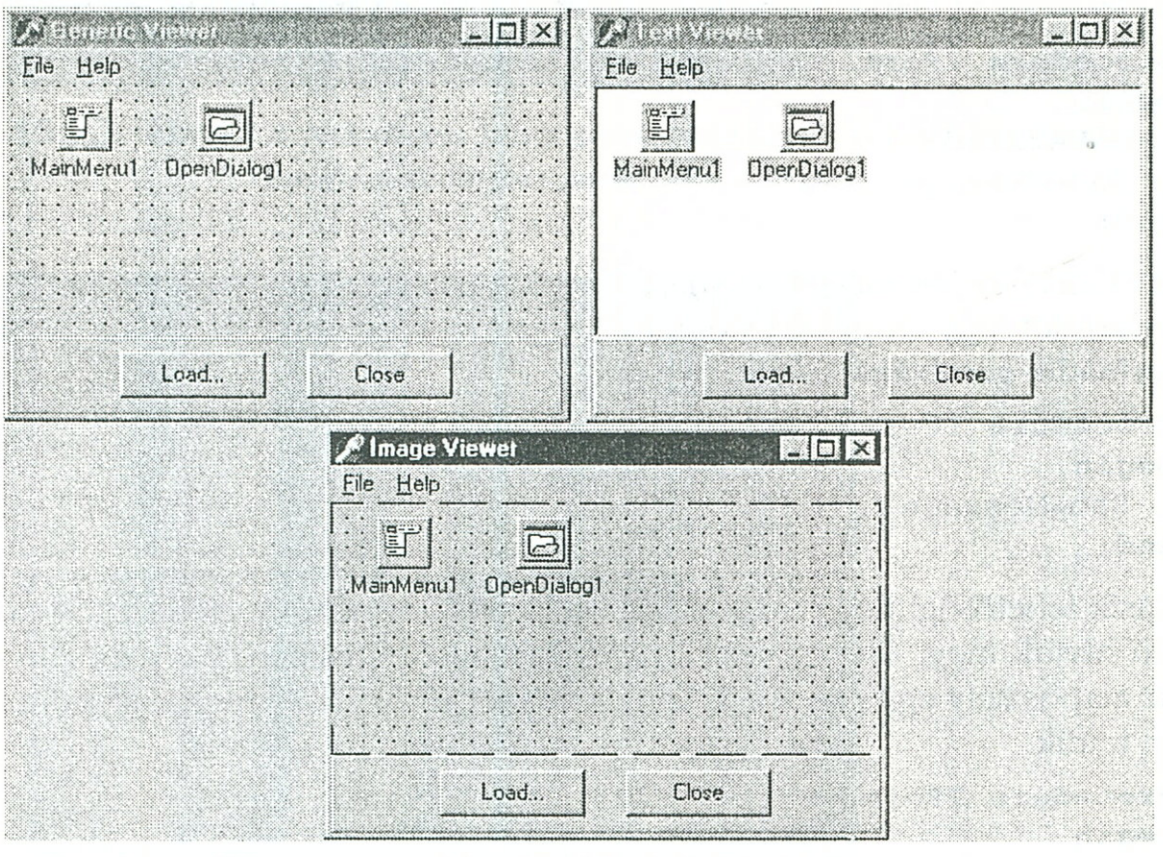
```
procedure TForm2.Button4Click(Sender: TObject);
begin
  inherited Button3Click (Sender);
  inherited;
end;
```

Ez ugyan nem túl általános, de jó tudni, hogy megvan rá a lehetőség. Természetesen az ős osztály bármely metódusát a saját formunk metódusának tekinthetjük, és szabadon meghívhatjuk. Ez a példa segít a vizuális form öröklődés néhány jellemzőjének megismerésében, de hogy a valódi erejét is láthassuk, egy összetettebb valós életbeli problémára vagy szükség, valami olyanra, amit itt nem láthatunk. Most azonban nézzünk valami mást: a vizuális form polimorfizmust.

Polimorf formok

A probléma egyszerű; ha egy formon definiálunk egy eseménykezelőt, amit egy leszármazott formon megváltoztatunk, akkor nem hivatkozhatunk a két metódusra az alaposztály valamely általános metódusával, mert az eseménykezelők alapértelmezés szerint statikus kötést használnak.

Zavaros? Akkor itt egy példa. Tegyük fel, hogy ugyanabban a programban szükségünk van egy szöveget és egy képet megjelenítő formra. A két form ugyanazokat az összetevőket tartalmazza, ugyanazt az eszköztárat, ugyanazt a menüt, OpenDialog komponenst, és néhány különböző komponenst, amit az adatok megjelenítését végzik. Ekkor dönthetünk úgy, elkészítünk egy alapformot, ami a két form közös összetevőit tartalmazza. A három formot tervezés közben láthatjuk a 13.24. ábrán.



13.24. ábra *A PoliForm példa alapformja, és a két leszármazott form tervezés közben*

Itt pedig a főform szöveges leírása következik:

```
object ViewerForm: TViewerForm
  Caption = 'Generic Viewer'
  Menu = MainMenu1
  object Panel1: TPanel
    Align = alBottom
    object ButtonLoad: TButton...
    object CloseButton: TButton...
  end
  object MainMenu1: TMainMenu
    object File1: TMenuItem...
      object Load1: TMenuItem...
      object N1: TMenuItem...
      object Close1: TMenuItem...
    object Help1: TMenuItem...
      object AboutPoliform1: TMenuItem...
  end
  object OpenDialog1: TOpenDialog...
end
```

A két leszármazott form csak csekély mértékben tér el, van rajtuk új komponens, egy képnéző (TImage), illetve egy szövegmegjelenítő (TMemo):

```
inherited ImageViewerForm: TImageViewerForm
  Caption = 'Image Viewer'
  object Image1: TImage [0]
    Align = alClient
  end
  inherited OpenDialog1: TOpenDialog
    Filter = 'Bitmap file|*.bmp|Any file|*.*'
  end
end

inherited TextViewerForm: TTextViewerForm
  Caption = 'Text Viewer'
  object Memo1: TMemo [1]
    Align = alClient
  end
  inherited OpenDialog1: TOpenDialog
    Filter = 'Text files|*.txt|Any file|*.*'
  end
end
```

A főform tartalmaz egy rövid általános kódot. A Close gomb, és a File menü Close parancsa meghívja a form Close metódusát. A Help menü About parancsa egy egyszerű üzenet ablakot jelenít meg. Az alap form Load gombja a következő kódot tartalmazza:

```
procedure TViewerForm.ButtonLoadClick(Sender: TObject);
begin
  ShowMessage ('Error: File loading code missing');
end;
```

A File menü Load parancsa azonban másik egy metódust hív meg:

```
procedure TViewerForm.Load1Click(Sender: TObject);
begin
  LoadFile;
end;
```

Ezt a metódust a TViewerForm osztályban a következőképpen deklaráltuk:

```
public
  procedure LoadFile; virtual; abstract;
```

Mivel ez egy absztrakt metódus, ezért a leszármazott formokon felül *kell* definiálnunk:

```
type
  TImageViewerForm = class(TViewerForm)
    Image1: TImage;
    procedure ButtonLoadClick(Sender: TObject);
  public
    procedure LoadFile; override;
  end;
```

Ez a LoadFile metódus az OpenDialog1 komponens segítségével felkéri a felhasználót, hogy válasszon ki egy képet, amit aztán betölt a kép komponensbe:

```
procedure TImageViewerForm.LoadFile;
begin
  if OpenDialog1.Execute then
    Image1.Picture.LoadFromFile (
      OpenDialog1.Filename);
end;
```

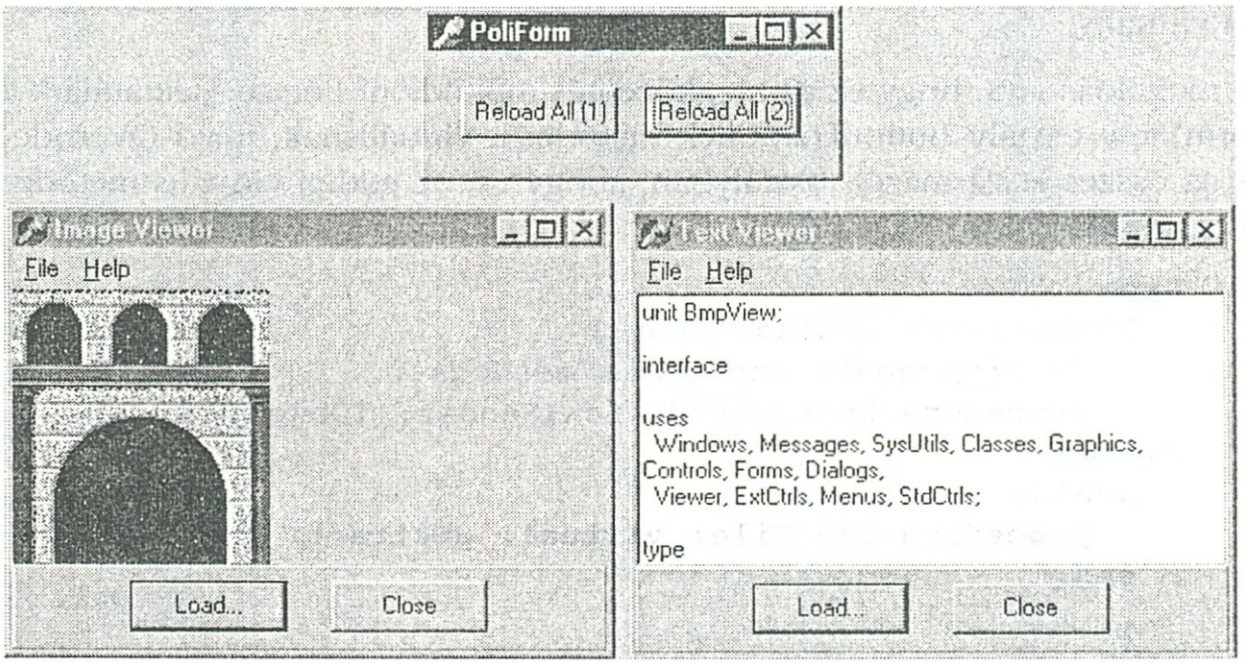
A másik osztály hasonló kóddal tölti be a szöveget a megfelelő komponensbe. A projektnek van még egy formja, a főform, amin két gomb van, ami két formon új állományokat olvassunk be. A főform az egyetlen ami a projekt indításakor létrejön. Az általános megjelenítő formot soha nem hozzuk létre, az csak alaposztályként szolgált a két leszármazott osztály számára, az általános kódot, és komponenseket tartalmazta. A két leszármazott form a főform OnCreate eseménykezelőjében jön létre:

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  FormList [1] := TTextViewerForm.Create(Application);
```

```
    FormList [2] := TImageViewerForm.Create(Application);
    for I := 1 to 2 do
      FormList[I].Show;
end;
```

A 13.25. ábrán láthatjuk a végleges formokat, már betöltött képpel és szöveggel. A FormList (formlista) egy polimorf form tömb, amit a TmainForm osztályban deklaráltunk:

```
private
  FormList: array [1..2] of TViewerForm;
```



13.25. ábra *A PoliForm példa futás közben*

Ahhoz, hogy ezt a deklarációt megtehessük, a főform interfész részének `uses` kifejezésébe be kell tennünk Viewer egységet (de a két leszármazott formot nem). A formtömböt használjuk arra, hogy a gombok megnyomásakor új állományokat olvassunk be. A két gomb OnClick esemenykezelője különböző megközelítést használ:

```
procedure TMainForm.ReloadButton1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 2 do
    FormList.ButtonLoadClick (self);
end;
```

```
procedure TMainForm.ReloadButton2Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 2 do
    FormList.LoadFile;
end;
```

A második gomb egyszerűen meghívja a virtuális metódust, és minden további nélkül működik. Az első gomb az eseménykezelőt hívja meg, és emiatt mindig az általános TFormView osztályt éri el, a ButtonLoadClick metódus pedig megjeleníti a hibaüzenetet. Ez azért következik be, mert ez a metódus statikus, és nem virtuális.

Van megoldás arra, hogy ez az megközelítés működjön? Persze: deklaráljuk a TFormView osztály ButtonLoadClick metódusát virtuálisnak, majd Overridenak az összes leszármazott osztályban, ahogy azt az eddigi virtuális metódusokkal tettük:

```
type
  TViewerForm = class(TForm)
    // components and plain methods...
    procedure ButtonLoadClick(Sender: TObject);
virtual;
  public
    procedure LoadFile; virtual; abstract;
  end;
...
type
  TImageViewerForm = class(TViewerForm)
    Image1: TImage;
    procedure ButtonLoadClick(Sender: TObject);
override;
  public
    procedure LoadFile; override;
  end;
```

Egyszerű, nem? Ez a trükk tényleg működik, bár a Borland dokumentációban erről szó sincs. A virtuális eseménykezelés az, amit vizuális form polimorfizmusnak nevezünk.

Hogyan tovább?

Ebben a fejezetben több módszert ismertünk meg arra, hogy hogyan készíthetünk olyan alkalmazásokat, amikben több mint egy form van. Láttuk, hogyan készíthetünk másodlagos modális vagy nem modális formokat, vagy modális párbeszédablakokat. A legfontosabb cél az volt, hogy megjegyezzük, hogy Delphiben a párbeszédablakok közönséges formok, különleges kerettel, és néhány más tulajdonsággal, de semmi egyéb.

Az alap példák mellett összetettebb dolgokkal is foglalkoztunk, mint a formok dinamikus létrehozása, bővíthető párbeszédablakok és a vizuális form öröklés. Megismertük az általános párbeszédablakokat, és a Delphi üzenet ablakokat, készítettünk Névjegyet, rejtett szöveggel, és nyitóképernyőként is felhasználtuk.

Másik lehetséges fejlesztés lehet az egyre elterjedtebb jegyzetfüzet metafora használata, ami a következő fejezet egyik témája. A legtöbb Window alkalmazás Options ablaka ezt a jegyzetfüzet felépítést követi. Másik lehetőség több ablakos programoknál az MDI alkalmazás, amiről a 15. fejezetben lesz szó. Az adatbázisokkal foglalkozó fejezetben beszélni fogunk még a tábla több ablakban történő összehangolt megjelenítéséről.

14. FEJEZET

Görgetés, többlapos formok

E fejezet tartalma:

- Amikor a formok túl nagyok
- A form görgetése
- Jegyzettöbmök készítése a Delphivel
- Jegyzettöbmök fülek nélkül és fülek jegyzettömbök nélkül
- Egy képkezelő program
- Formfelosztási technikák
- Alternatív elválasztó komponensek

Görgetés, többlapos formok

Kezdetben a Windows alkalmazások felhasználói felülete nagyon egyszerű volt. Ahogy teltek az évek, a legelterjedtebb alkalmazások és rendszerek új szolgáltatásai fejlesztették ezt. Például, a használhatóság növelése érdekében került be az eszköztár, a státuszsor is. Másik bevált ötlet az egyszerű utánzó szóképek, amelyek a mindennapokban használt eszközökön alapulnak; ilyen szókép egy jegyzettömb fülekkel, amikkel mozoghatunk különféle oldalak és területek között. A Delphi számos ilyen elemet tartalmaz. A füleket használják az Options párbeszédablakban, az editorban, az Object Browserben és a komponenspalettán.

Számos alkalmazásnak a felhasználói felülete hasonlóan néz ki. Néhány alkalmazás, beleértve a vezető táblázatkezelőket is, szintén használja a jegyzettömb szóképet a saját fő ablakaikban. A jegyzettömbök és fülek használata olyan fontossá vált, hogy a Microsoft közvetlen támogatást adott neki a Windows 95-ben.

De mielőtt belemélyednénk a jegyzettömbökbe és fülekbe, bemutatjuk egy egyszerűbb, de hatékony technika alkalmazását ahhoz, ha szükség van több komponens formba tömörítéséhez: a görgetést.

Amikor a formok túl nagyok

Amikor egy egyszerű alkalmazást készítünk, egy form tartalmazhatja az összes szükséges komponenst. Amint az alkalmazások növekednek, szükség lehet a komponensek összepréselésére, a form méretének növelésére, vagy új formok hozzáadására. Ha csökkentjük a komponensek által elfoglalt helyet, beépíthetünk olyan képességet, hogy futáskor újraméretezze azt, végső fokon szétválassza a formot különböző területekre. Ha a form méretének növelését választjuk, szükséges lehet a görgetősávok használata, amik lehetővé teszik a felhasználónak a mozgást végig az egész, a képernyőnél nagyobb formban.

Végül, ha az új form hozzáadását választjuk, háromféleképpen járhatunk el:

- Másodlagos formokat és párbeszédablakokat készítünk, a 13. fejezetben leírtak szerint.
- A Windows 95 PageControl, vagy a Delphi NoteBook komponensét használjuk a többboldalas formok létrehozásához, az ebben a fejezetben leírtaknak megfelelően.
- Követjük a tipikus Windows MDI megjelenítést, amit a 15. fejezetben ismertetünk.

Többboldalas formokat a Windows alkalmazások párbeszédablakaiban körülbelül a Microsoft Winword 2.0 és a Quattro Pro for Windows óta használnak.

A legújabb generációs Microsoft alkalmazások egy újfajta megjelenítésű jegyzettömböt mutattak be a párbeszédablakokban, fülekkel az oldalukon. Ez általánossá vált a Windows 95-ben, bár a Windows 95 fülvezérlőjének felhasználói felülete kissé különbözik elődjétől.

A Delphi 2-nek mind TabControl, mind PageControl komponense van, ami a Windows 95 általános vezérlőire épül, és egy egyedi Notepad (jegyzettömb) komponens, amely kombinálható egy egyedi fül komponenssel, hogy füles jegyzettömböket hozzunk létre. A jegyzettömb komponens mindemellett használható más elemekkel együtt is, amint ebben a fejezetben később látni fogjuk.

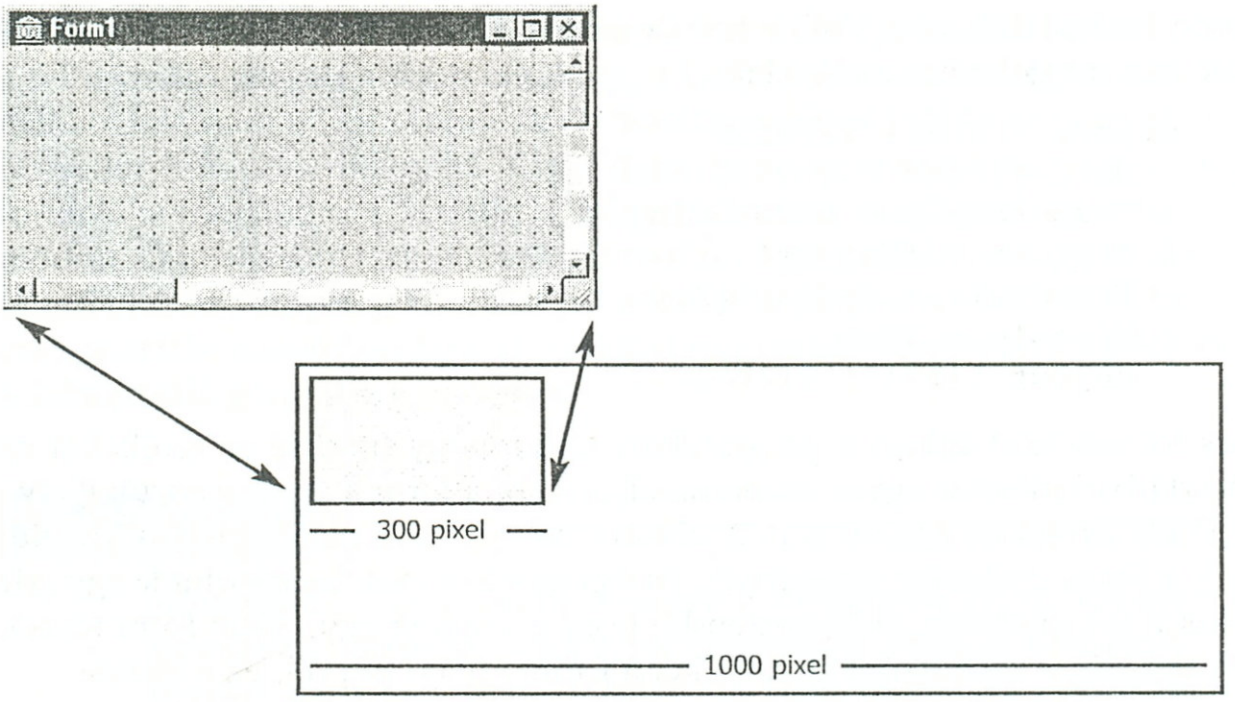
A form görgetése

Egy görgetősáv hozzáadása egy formhoz egyszerű. Valójában nem kell semmit sem csinálni. Ha sok komponenst rakunk egy nagy formba és utána csökkentjük a méretét, egy görgetősáv adódik automatikusan a formhoz egészen addig, amíg meg nem változtatjuk az AutoScroll tulajdonság értékét, ami alapbeállításban True.

Az AutoScroll mellett a formoknak két tulajdonságuk van, HorzScrollBar és VertScrollBar, amiket számos tulajdonság beállításához használhatunk a formhoz hozzárendelt két TFormScrollBar objektumban. A Visible (láthatóság) tulajdonság megmutatja, ha a görgetősáv jelen van, a Position (pozíció) tulajdonság meghatározza a görgető gomb kezdeti helyzetét, és az Increment (növekmény) tulajdonság meghatározza a görgetősáv két végén lévő nyilakra kattintás hatását. A legfontosabb tulajdonság mindamellett a Range (méret).

Egy görgetősáv Range tulajdonsága adja meg a form egy irányú virtuális méretét, nem pedig a görgetősáv értékeinek aktuális mérete. Elsőre ez zavaró lehet. Íme egy példa, hogy tisztázzuk, a Range tulajdonság hogyan is működik. Tegyük fel, hogy szükségünk van egy sok komponenssel rendelkező formra, és így a formnak 1000 pixel szélesnek kell lennie. Használhatjuk ezt az értéket a form virtuális méretének beállításához, megváltoztatva a vízszintes görgetősáv méretét. Nézzük meg a 14.1 ábrát, amin illusztráltuk egy form virtuális méretét beleértve egy görgetősáv méretét. Ha a form kliens területének szélessége kisebb, mint 1000 pixel, egy görgetősáv jelenik meg. Most el lehet kezdeni használni ezt a tervezésnél, hogy új komponenseket adjunk a form rejtett részeihez.

A görgetősáv Position tulajdonsága 0 és 1000 érték közötti értéket vehet fel, levonva belőle a kliens terület aktuális méretét. Valójában ha a form kliens területe 300 pixel széles, akkor 700 pixel görgethető, hogy elérjük a form túlsó végét (az ezredik pixelt).



14.1. ábra *Egy form virtuális méretének illusztrációja egy görgetősáv határaival*

A görgetést tesztelő példa

Egy Scroll1 nevű mintaprogramot készítettünk, aminek 1000 pixel széles virtuális formja van. Hogy ezt elérjük, egyszerűen beállítottuk a vízszintes görgetősáv méretét 1000-re:

```
object Form1: TForm1
  Width = 458
  Height = 368
  HorzScrollBar.Range = 1000
  VertScrollBar.Range = 305
  AutoScroll = False
  Caption = 'Scrolling Form'
  OnResize = FormResize
  ...
```

A példában szereplő formot kitöltöttük sok, jelentés nélküli listapanellel, és el tudtuk érni ugyanazt a görgetősáv méretet azáltal, hogy behelyeztük a legjobboldalibb listapanelt úgy, hogy ennek pozíciója (Left) plusz ennek mérete (Width) megegyezzék 1000-rel.

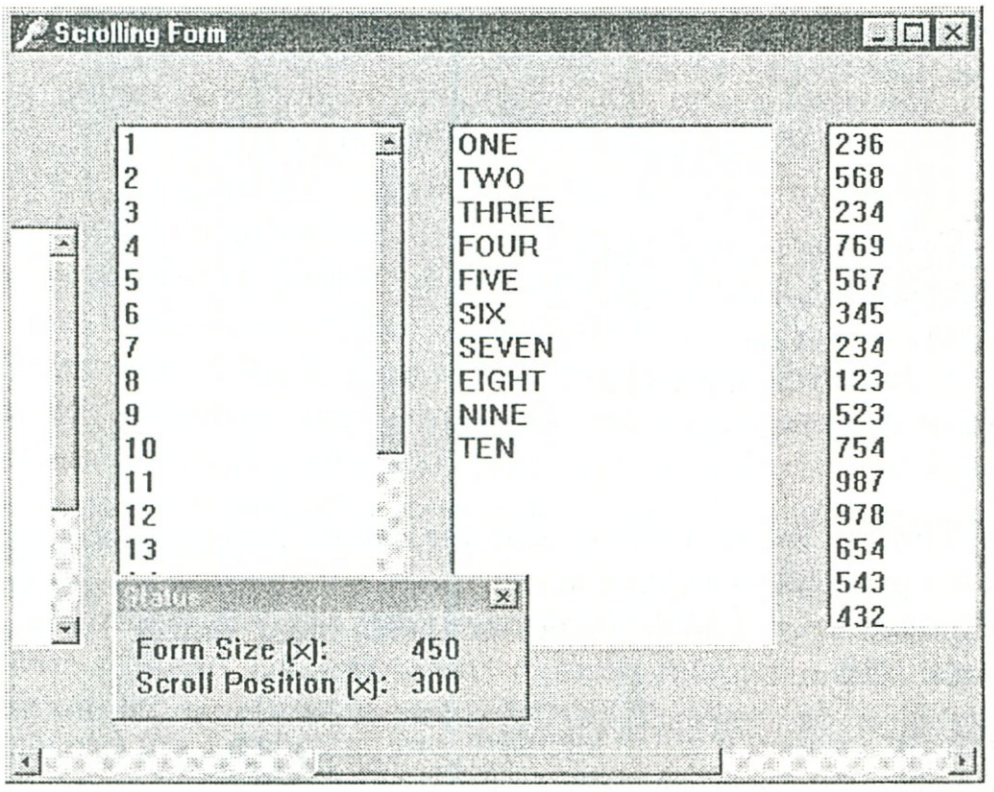
A példa érdekes része az eszköztár ablak jelenléte, ami mutatja a form és a vízszintes görgetősáv helyzetét. Ezen második formnak 4 címkéje van, kettő fix szöveggel és kettő az aktuális kimenettel. Mindemellett, a másodlagos formnak (Status) van egy bsToolWindow keretstílusa; ez az egyik legfontosabb ablak.

Szintén be kell állítani a Visible tulajdonságát True-ra, hogy induláskor automatikusan megjelenítse az ablakát:

```
object Status: TStatus
  BorderIcons = [biSystemMenu]
  BorderStyle = bsToolWindow
  Caption = 'Status'
  FormStyle = fsStayOnTop
  Visible = True
  object Label1: TLabel...
```

Nincs túl sok kód ebben a programban. Célja, hogy frissítse az értékeket az eszköztárban minden egyes alkalommal, amikor a formot átméretezzük illetve görgetjük (amint ez látható a 14.2. ábrán). Az első rész nagyon egyszerű. Kezeljük a form OnResize eseményét, és egyszerűen csak bemásolunk egy pár értéket a két címkébe. Megjegyzendő, hogy a címkék egy másik form részei, ezért szükséges előtagként ellátni őket a formra utaló névvel, ez a Status:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  Status.Label3.Caption := IntToStr(ClientWidth);
  Status.Label4.Caption :=
    IntToStr(HorzScrollBar.Position);
end;
```



14.2. ábra *A Scroll1 példa kimenete*

Ha meg szeretnénk változtatni a kimenetet minden egyes alkalommal, amikor a felhasználó görgeti a form tartalmát, a Delphi eseménykezelőjét nem használhatnánk, mert nincs a formnak OnScroll eseménye (habár van egy az önálló ScrollBar komponenseknek). Ésszerű kihagyni az eseményt, mert a Delphi formok automatikusan kezelik a görgetősávokat, méghozzá igen hatékony módon. Összehasonlításként, a Windowsban a görgetősávok nagyon alacsony szintű elemek, rengeteg kódolást igényelnek. A görgetési események kezelése csak speciális esetekben ésszerű, például ha precízen nyomon akarjuk követni a felhasználó görgetési manővereit.

Megjegyzés

Mégegyszer, amit igazán szeretek a Delphiben az az, ahogyan egy Windows üzenetet, amit nem támogatott idáig a környezet, csak eggyel több sort igényel, hogy használjuk. Semmi ehhez foghatót nem láttam más vizuális környezetben.

Itt a kód, amit meg kell írnunk. Először, adjunk az osztályhoz egy eljárás deklarációt, és rendeljük egymáshoz a Windows vízszintes gördülő üzenetével (wm_HScroll):

```
public
    procedure FormScroll (var ScrollData: TWMScroll);
      message wm_HScroll;
```

Ezután írjuk meg az eljárás kódját, ami majdnem ugyanaz, mint az előbb látott FormResize eljárás:

```
procedure TForm1.FormScroll(var ScrollData: TWMScroll);
begin
  inherited;
  Status.Label3.Caption := IntToStr(ClientWidth);
  Status.Label4.Caption :=
    IntToStr(HorzScrollBar.Position);
end;
```

Fontos, hogy az inherited hívást is használjuk, amely megteremti az összeköttetést az eljárás és az alap osztályú formban lévő üzenet között. Megjegyezzük, hogy az öröklött kulcsszó a Windows üzenet leírójában meghívja azt az alap osztályú metódust, amit felülírunk, amelyik pont a megfelelő Windows üzenettel van kapcsolatban (még ha az eljárás neve különböző is). Enélkül a hívás nélkül a formnak nem lenne meg az alapértelmezésbeli görgetési viselkedése, ami annyit jelent, hogy egyáltalán nem görgetne.

Automatikus görgetés

A görgetősáv Range tulajdonsága furcsa dolognak tűnhet, amíg el nem kezdjük következetesen használni. Ha jobban belegondolunk, meg fogjuk érteni ennek a megközelítésnek az előnyeit. Mindenekelőtt, a görgetősáv automatikusan eltűnik a formból, ha a form kliens területe elég nagy ahhoz, hogy beleférjen a virtuális méret; amikor pedig csökkentjük a form méretét, a görgetősáv újból megjelenik.

Ez a tulajdonság különösen érdekessé válik akkor, ha a form AutoScroll tulajdonsága True-ra van állítva. Ebben az esetben, a legjobboldalibb és alsóbb vezérlők extrém pozíciói automatikusan átmásolódnak a form két görgető sávjának Range tulajdonságaiba. Az automatikus görgetés jól működik a Delphiben. Az utolsó példánkban a form virtuális méretét az utolsó listaelem jobb oldalára állítanánk. Ezt a következképpen tehetjük:

```
object ListBox6: TListBox
  Left = 832
  Width = 145
end
```

Ezért a form vízszintes virtuális mérete 977 lenne (ami a két fenti érték összege). Ez a szám automatikusan bemásolódik a form HScrollBar tulajdonságának Range mezejébe, amíg manuálisan meg nem változtatjuk azt egy nagyobb form létrehozásához (amint a Scroll1 példában is tettük, 1000-re állítva, hogy hagyjuk némi helyet az utolsó listaelem és a form kerete között). Ez az érték látható az Object Inspectorban, vagy nézzük az alábbi tesztet: Futtassuk a programot, méretezzük a formot tetszőlegesen, és mozgassuk a görgető gombot a jobb szélső pozícióba. Amikor összeadjuk a form méretét és a gomb pozícióját, a mérettől függetlenül mindig 1000-et kapunk, ami a form jobb szélső pixelélnek a virtuális koordinátája.

Kép görgetése

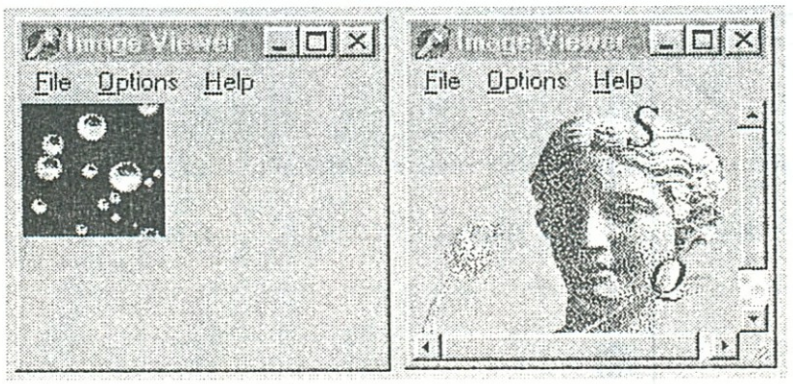
A Delphiben működő automatikus görgetés egyik előnye az, hogy ha egy formban levő önálló nagy komponens mérete megváltozik, akkor automatikus görgetősáv jelenik meg illetve tűnik el. Jó példa erre az Image (kép) komponens használata. Ha eme komponens AutoSize tulajdonsága True, és új képet töltünk bele, akkor a komponens önmagát fogja méretezni, és a form vagy hozzáadja, vagy eltünteti a görgetősávokat.

Egy példa talán segíthet megérteni, hogyan működik a kép görgetése. Emlékezzünk a 11. fejezetben készített képnézegető programunkra: a formja egy fájlból betöltött bitmapet jelenített meg, vagy megnövelve a méretét a form méretére, vagy az eredeti méretben. Az a probléma, hogy ha az eredeti méret túl nagy ahhoz, hogy beleférjen a formba, akkor a bitmap egy része rejtve marad.

A probléma megoldása igen egyszerű. Be kell állítani egy kis kezdeti méretet is a képnek. Nem kell semmi méretezést sem csinálni új kép betöltésekor, hiszen ezt a rendszer automatikusan megteszi helyettünk. Írjuk be az alábbi kódot:

```
procedure TViewerForm.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    Image1.Picture.LoadFromFile (OpenDialog1.FileName);
end;
```

A 14.3. ábrán látható, hogy a görgetősávok hozzáadódtak a formhoz. Az ábra két különböző változatát mutatja a programnak. A különbség a bal és jobb oldali program között az, hogy az elsőnek a képe kisebb, mint a kliens terület, ezért görgetősávok nem jelentek meg. Amikor egy nagyobb képet töltünk be a programban, két görgetősáv jelenik meg automatikusan, amint a jobb oldali példán látható.



14.3. ábra *Az ImageV2 példában a görgetősávok automatikusan létrejönnek a formban, amikor a teljes bitmap nem fér bele a form kliensterületére*

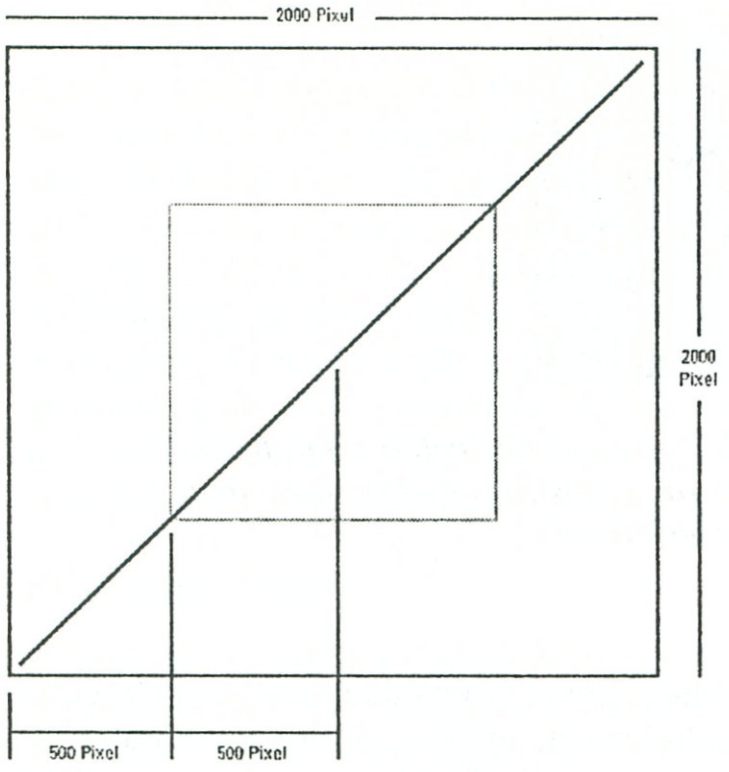
Valamivel több programozás szükséges ahhoz, hogy kikapcsoljuk a görgetősávokat illetve megváltoztassuk a kép helyzetét, amikor a Strech menüparancsot kiválasztjuk, és visszaállítsuk, amikor ez a tulajdonság nem érvényes. Tehát közvetlenül nem állítjuk a görgetősávokat, hanem egyszerűen megváltoztatjuk a panel helyzetét a Strech tulajdonságot használva, és kézzel kiszámítjuk az új méretet az aktuális betöltött kép méretét használva fel (ez a kód utánozza az AutoSize tulajdonság hatását, ami csak akkor működik, ha új fájlt töltünk be):

```
procedure TViewerForm.Stretch1Click(Sender: TObject);
begin
  Image1.Stretch := not Image1.Stretch;
  Stretch1.Checked := Image1.Stretch;
```

```
    if Image1.Stretch then
      Image1.Align := alClient
    else
    begin
     Image1.Align := alNone;
     Image1.Height := Image1.Picture.Height;
     Image1.Width := Image1.Picture.Width;
    end;
  end;
```

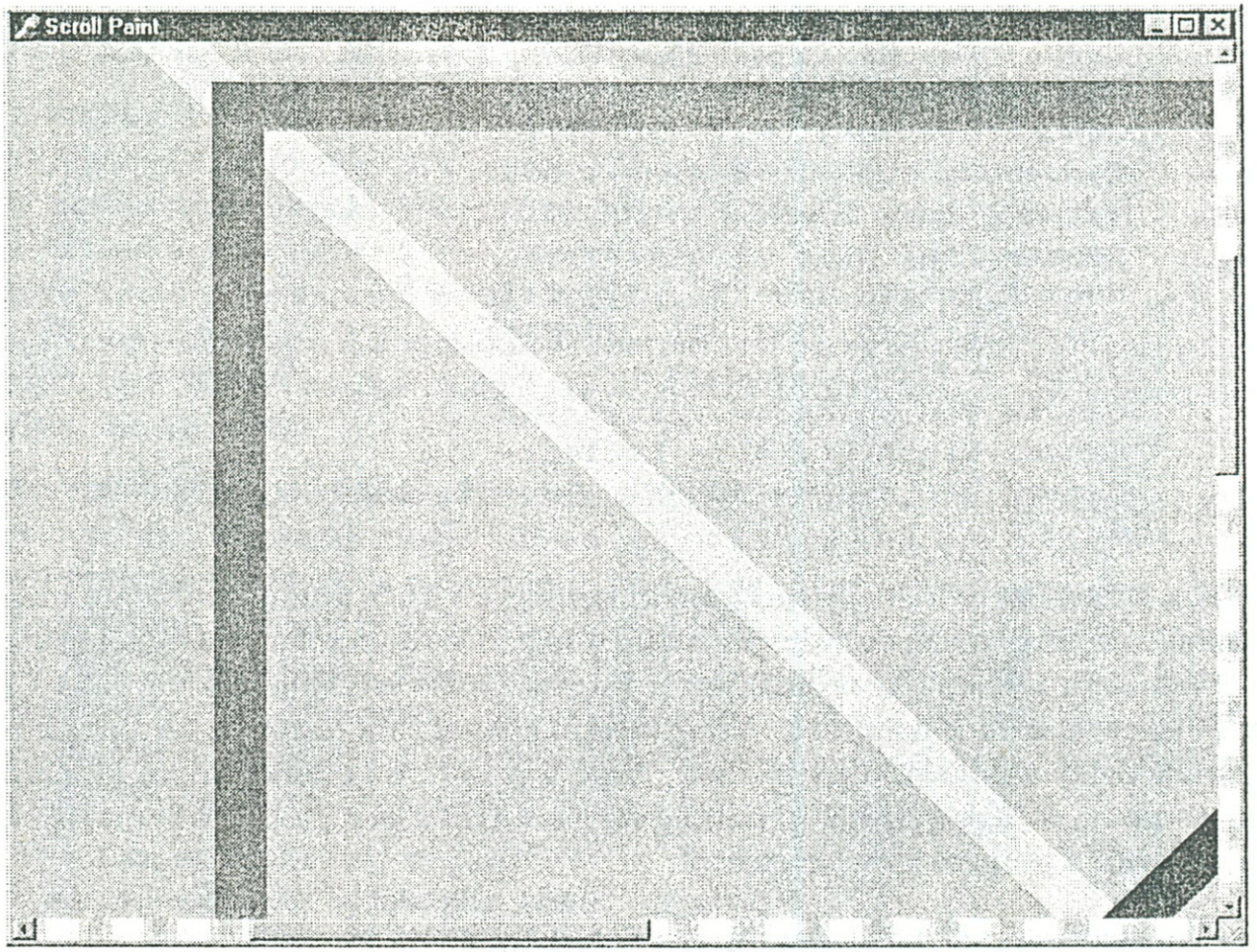
Görgetés és form koordináták

Amint az előző két rész példáiban láttuk, a formok automatikusan képesek a komponenseik görgetésére. De mi történik akkor, ha közvetlenül a form felületén festünk? Egy kis probléma adódik, de a probléma kézben tartható. Tegyük fel, hogy néhány vonalat szeretnénk rajzolni egy form virtuális felületén, amint a 14.4. ábra mutatja.



14.4. ábra *Egy form virtuális felületére rajzolandó vonalak*

Mivel feltehetőleg nincsen mindkét irányban 2000 pixelt kijelezni képes monitorunk, csinálhatunk egy kisebb formot, adjunk hozzá két görgetősávot, állítsuk be azok méretét (Range), amint a Scroll2 példában tettük. A form szöveges leírása:



14.5. ábra *A Scroll2 példa egy lehetséges kimenete*

```
object Form1: TForm1
  HorzScrollBar.Range = 2000
  VertScrollBar.Range = 2000
  ClientHeight = 336
  ClientWidth = 472
  OnPaint = FormPaint
  end
```

Ha csak egyszerűen megrajzoljuk a vonalakat, a form virtuális koordinátáit használva, akkor a kép nem lesz tökéletes. Valójában az OnPaint válaszmetódusban ki kell számolnunk magukat a virtuális koordinátákat. Szerencsére ez egyszerű, mivel tudjuk, hogy a virtuális X1 és Y1 koordinátája a kliens terület bal felső sarkának megegyezik a két görgetősáv aktuális pozíciójával:

```
procedure TForm1.FormPaint(Sender: TObject);
var
  X1, Y1: Integer;
begin
  X1 := HorzScrollBar.Position;
  Y1 := VertScrollBar.Position;
```

```
    {draw a yellow line}
    Canvas.Pen.Width := 30;
    Canvas.Pen.Color := clYellow;
    Canvas.MoveTo (30-X1, 30-Y1);
    Canvas.LineTo (1970-X1, 1970-Y1);
    {draw a blue line}
    Canvas.Pen.Color := clNavy;
    Canvas.MoveTo (30-X1, 1970-Y1);
    Canvas.LineTo (1970-X1, 30-Y1);
    {draw a fuchsia square}
    Canvas.Pen.Color := clFuchsia;
    Canvas.Brush.Style := bsClear;
    Canvas.Rectangle (500-X1, 500-Y1, 1500-X1, 1500-Y1);
end;
```

Szintén láthatunk példát egy program kimenetére a 14.5. ábrán. Próbáljuk meg használni a programot, és megváltoztatni a rajzolási funkciókat, de ne felejtsük el alkalmazni a relatív koordinátákat a form virtuális eredetijére a virtuális X1 és Y1 koordináták kivonásával. Azt is kipróbálhatjuk, mi történik, ha sima koordinátákat használunk. Úgy találjuk, hogy a program kimenete nem jó, nem fog görgetni, és ugyanaz a kép mindig ugyanabban a pozícióban fog maradni, függetlenül a görgetési műveletektől.

Jegyzettömbök készítése a Delphivel

Három út kínálkozik fülekkel rendelkező jegyzettömbök létrehozására, hogy az aktuális oldalt megváltoztassuk:

- Használjuk a Windows 95 PageControl (Lapvezérlő) komponensét.
- Használjuk a Notebook (Jegyzettömb) komponenst a TabSet (Fülbeállító) komponenssel együtt.
- Használjuk a TabbedNotebook (Füles jegyzettömb) komponenst, amit a Delphi 32 bites verziója még tartalmaz, de csak a kompatibilitás miatt.

A három mód közötti fő különbség ezen komponensek felhasználói felülete között van, amint a következő rész ábráin látható. Általánosan, amióta a Lapvezérlő és Fülvezérlő komponensek a Windows általános vezérlőjén alapszanak, valószínűleg használnunk kell, kivéve, ha kompatibilitási problémák vagy egyéb speciális okok ezt nem engedik meg. Technikailag a fülvezérlő kicsit több feladatot lát el, mint hogy megjelenítse a saját füleit. A TPageControl osztály egy fülvezérlőn plusz néhány panelen (TabSheet-ek) alapszik, és több lehetőséget nyújt.

Ha csak egy fülre van szükségünk a jegyzettömb illetve lapok nélkül, használhatjuk a TabControl vagy TabSet komponenst. Ha az oldalak kellenek fülek nélkül, használhatjuk a PageControlt (fülek elrejtése) vagy a Notebook (Jegyzettömb) komponenst saját magát.

PageControl és TabSheet komponensek

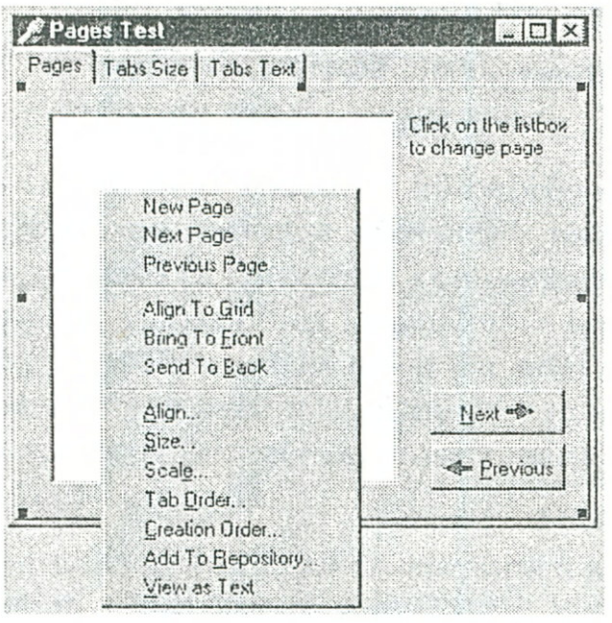
A Windows 95-ben három komponens van, melyekkel az oldalakat és füleket kezeljük, Delphiben a TabControl (Fülvezérlő) és PageControl (Oldalvezérlő) komponensek állnak rendelkezésünkre, illetve a PageControlon belül a TabSheet komponens. A TabControl arra való, hogy önálló füleket hozzunk létre (amik nem kapcsolódnak oldalakhoz; ezt később használni fogjuk egy speciális példában). A PageControl komponens pedig applikáció illetve párbeszédablak létrehozására szolgál az általános többboldalas vagy jegyzettömb eljáráson keresztül.

Amint általában, a súgórendszer tulajdonság-listájának és a PageControl komponens metódusainak megkettőzése helyett készítettünk egy példát, ami kibővíti ezek képességeit és megengedi, hogy futás alatt változtassuk meg a tulajdonságaikat. A Pages nevű példa egy 3 oldalas PageControllal rendelkezik. Ha ezt létre akarjuk hozni, egyszerűen be kell helyezni a vezérlőt egy új formba, ki kell választani és használni kell a lokális menüt, hogy létrehozzunk újabb oldalakat vagy füloldalakat. Ezután adjunk komponenseket hozzá és állítsuk be a tulajdonságaikat a következő szöveges leírású példa alapján:

```
object Form1: TForm1
  Caption = 'Pages Test'
  OnCreate = FormCreate
  object PageControl1: TPageControl
    ActivePage = TabSheet1
    Align = alClient
    object TabSheet1: TTabSheet
      Caption = 'Pages'
      object Label3: TLabel...
      object ListBox1: TListBox...
    end
    object TabSheet2: TTabSheet
      Caption = 'Tabs Size'
      object Label1: TLabel...
      object EditWidth: TEdit...
      object UpDown1: TUpDown
        Associate = EditWidth
        Min = 0
        Max = 2000
        Position = 0
      end
      object Label2: TLabel...
      object EditHeight: TEdit
        Text = '0'
      end
      object UpDown2: TUpDown
        Associate = EditHeight
        Min = 0
```

```
        Max = 2000
        Position = 0
      end
      object CheckBoxMultiLine: TCheckBox...
      object CheckBoxVisible: TCheckBox...
        Caption = '&Last Tabs &Visible'
        State = cbChecked
      object BitBtnApply: TBitBtn...
    end
    object TabSheet3: TTabSheet
      Caption = 'Tabs Text'
      object Memo1: TMemo...
      object BitBtnChange: TBitBtn
        Caption = '&Change'
        Enabled = False
        OnClick = BitBtnChangeClick
      end
    end
  end
  object BitBtnPrevious: TBitBtn...
  object BitBtnNext: TBitBtn...
end
```

Minden TabSheet objektumnak van saját címkéje (Caption), ami az oldal fülén jelenik meg. Az oldalakak megváltoztatását ugyanabban a lokális menüben tehetjük meg. Ugyanezt érhetjük el a fülekre, majd a form felületére kattintással a TabSheet kiválasztásához. A 14.6. ábrán látható a PageControl komponens lokális menüje, az első oldallal együtt. Ez az oldal tartalmaz egy listapanelt és egy kis címkét, illetve megosztva két gombot más oldalakkal.



14.6. ábra *A Pages és példa PageControljának első oldala, annak lokális menüjével*

Ha egy oldalra helyezünk egy komponenst, akkor az csak az adott oldalon érhető el. Hogyen érhetjük el ugyanazt a komponenst (ebben az esetben két bitmap gombbal) minden egyes oldalról anélkül, hogy lemásolnánk ? Egyszerűen csak rakjuk a komponenst a formra, a PageControlon kívül (vagy mielőtt hozzárendeljük a kliensterülethez) és mozgassuk az oldalak elé; ezt a form helyi menüjének **Bring to Front** (előtérbe hozás) parancsának hívjuk. A minden egyes oldalra helyezett két gomb használható az oldalak közötti előre- és hátralapozásra, illetve egy alternatíva a fülek használatára. Egyikükhöz rendelt kód a következő:

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
  PageControl1.SelectNextPage (True);
end;
```

A másik gomb ugyanazt az eljárást hívja, False paramétert adva át az előző oldal kiválasztásához. Vegyük észre, hogy nincs szükség ellenőrizni, az első vagy utolsó oldalon vagyunk-e, mert a SelectNextPage (következő oldal kiválasztása) metódus figyelembe veszi, hogy az utolsó oldal az első előtt lesz, és közvetlenül lapoz ezen két oldal között.

Most, hogy definiáltuk az általános komponens hsználatát, újra szemügyre vehetjük az első oldalt. Van egy listapanelje, ami futáskor tartalmazni fogja a fülek nevét. Ha a felhasználó rákattint a panel egyik elemére, az aktuális oldal megváltozik. Ez a harmadik módszer az oldalak váltására (a fülek és gombok után). A listapanel a FormCreate (formkészítő) metódusban van kitöltve, hozzárendelve a form OnCreate eseményével, másolja minden egyes oldal címkéjét (a Page tulajdonság tárol egy listát a TabSheet objektumokról):

```
for I := 0 to PageControl1.PageCount - 1 do
  ListBox1.Items.Add (PageControl1.Pages.Caption);
```

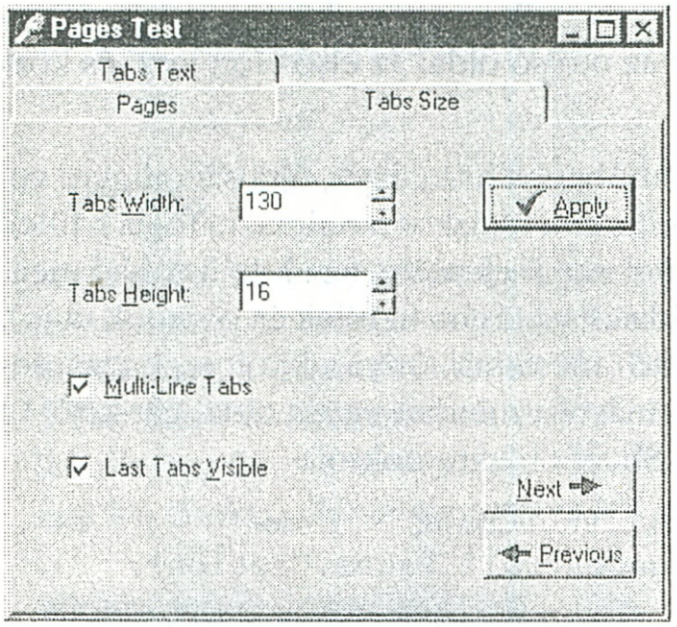
Amikor egy listaelemre kattintunk, kiválaszthatjuk a megfelelő oldalt:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  PageControl1.ActivePage :=
    PageControl1.Pages [ListBox1.ItemIndex];
end;
```

A második oldal két szerkesztő panelt tartalmaz (két UpDown komponenssel összekötve) és két jelölőnégyzetet, amint a 14.7. ábrán látható. A felhasználói input egy szám (vagy válaszuk ki a fel/le gombokra kattintással, illetve nyomjuk le a ↑ vagy ↓ nyilakat mialatt a megfelelő szerkesztő panelben vagyunk) és jelöljük ki a jelölőnégyzeteket, majd nyomjuk meg az Apply gombot, hogy a változás létrejöjjön:

```
procedure TForm1.BitBtnApplyClick(Sender: TObject);
begin
  {set tabs width, height, and lines}
  PageControl1.TabWidth := StrToInt (EditWidth.Text);
  PageControl1.TabHeight := StrToInt (EditHeight.Text);
  PageControl1.MultiLine := CheckBoxMultiLine.Checked;
  {show or hide the last tab}
  TabSheet3.TabVisible := CheckBoxVisible.Checked;
end;
```

Ezzel a kis kóddal meg tudjuk minden egyes fül szélességét és magasságát változtatni (emlékezzünk csak vissza, hogy a 0 azt jelenti, hogy a méret automatikusan számítódik a minden egyes string által elfoglalt helyből), és ki tudjuk választani hogy többsoronyi fület, vagy két kis nyilat görgessünk a fül területén. Az utolsó fület el is rejthetjük a PageControlon, ami megfelel a TabSheet3 komponensnek.



14.7. ábra *A példa második oldala a fülek méretezésére és pozicionálására használható: láthatók nagy fülek és többszörös sorok*

Ha a fülek egyikét elrejtjük a Tab Visible tulajdonságának False beállításával, akkor a fül nem elérhető a Next (következő) és Previous (előző) gombokra kattintással, ami a SelectNextPage metóduson alapszik. Ehelyett, a FindNextPage függvényt célszerű használni, amint az alábbi részlet mutatja a NextButton komponens OnClick esemény kezelőjének ebben az új verziójában:

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
  PageControl1.ActivePage :=
    PageControl1.FindNextPage (
      PageControl1.ActivePage, True, False);
end;
```

Figyelem!

Amikor megváltoztatjuk egy PageControl füleinek magasságát illetve többszörös vonalakat állítunk be futáskor, az oldalon minden komponens lefelé mozog, és ki is csúszhat a látható területről. Ugyanakkor ha vannak komponensek, amik nem a PageControl „gyermekei", de eléjük vannak rakva (mint a két bitmap gomb a Pages példában), azok nem fognak mozogni, sőt elrejthetik bizonyos oldalak más komponenseit.

Az utolsó oldalnak van egy feljegyzés komponense, megint az oldalak neveivel (hozzáadva a FormCreate metódusban). Szerkeszthető az oldalak neve, illetve a Change gomb lenyomásával a fülek szövege is, de csak ha a stringek száma megegyezik a fülek számával:

```
procedure TForm1.BitBtnChangeClick(Sender: TObject);
var
  I: Integer;
begin
  if Memo1.Lines.Count <> PageControl1.PageCount then
    MessageDlg ('One line per tab, please',
                mtError, [mbOK], 0)
  else
    for I := 0 to PageControl1.PageCount -1 do
      PageControl1.Pages [I].Caption :=
        Memo1.Lines [I];
  BitBtnChange.Enabled := False;
end;
```

Újabb tulajdonság amit adhatunk a példához egy módszer arra, hogy hozzáadhatunk futáskor új oldalakat és beléjük helyezhetünk új komponenseket. Szintén hozzáadhatunk egy legörgülő menüt az oldalak listázásához, hogy egy negyedik módszert is felajánljunk az oldalak változtatásához.

Tipp

Amikor egy, a PageControlra vagy jegyzettömbre alapuló formot írunk, emlékezzünk arra, hogy az első, futáskor megjelenített oldal az az oldal, amin a kód fordítása előtt voltunk. Ez azt jelenti, hogyha a harmadik oldalon dolgozunk, és ekkor lefordítjuk, és futtatjuk a programot, erről az oldalról fog indulni. Szokásos megoldása a problémának egy sor beszúrása a FormCreate metódusban, miszerint a PageControl vagy a jegyzettömb legyen az első oldal. Ezzel a tervezéskori aktuális oldal nem befolyásolja a futtatáskori kezdőoldalt.

Notebook TabSet-tel

A szokásos módszer fülekkel ellátott jegyzettömb létrehozásához a 16 bites Delphiben az volt, hogy két különálló komponenst használtunk, a Jegyzettömböt és a TabSet-et. Ez most is járható út, mert egyrészt rugalmas, másrészt a Delphi 16 bites és 32 bites verziója kompatibilis egymással. Egy, a PageControlt használó program valójában nem fordítható újra a 16 bites Delphiben. Hogy a Notebook és TabSet komponenseket részletezzük, egy, az előzőhöz hasonló példát írtunk Tab1 néven.

Ezen form létrehozásához a Notebook komponenst rakjuk a formra, majd rakjunk rá egy TabSet komponenst is, helyezzük a TabSetet a form aljára (alBottom), és helyezzük a Nootbookot a kliens területre (alClient). Most nevet adhatunk a jegyzettömb lapjainak a Pages tulajdonság kiválasztásával, ahol a megfelelő szerkesztőbe beírunk pár értéket. A következő lépés a fülek előkészítése azáltal, hogy beírunk pár stringet a Tabs tulajdonságba. Jó ötlet a TabSet komponenek füleinek és a jegyzettömb oldalainak ugyanazt a nevet adni. Mindamellett a TabSet komponensben használt nevek nem különösen fontosak, mivel nem láthatóak. A fontos az, hogy maguknak a füleknek legyen jelentéstartalommal bíró nevük, mert ezek láthatóak a felhasználó számára.

Hogy egy jegyzettömböt a fülkészlethez kapcsoljunk, legalább egy sort meg kell írnunk eleget téve a TabSet komponens OnChange eseményének:

```
procedure TForm1.TabSet1Change(Sender: TObject;
  NewTab: Integer; var AllowChange: Boolean);
begin
  Notebook1.PageIndex := NewTab;
end;
```

Tipp

Az OnChange válaszmetódusban a NewTab paramétert kell használnunk és nem a fül aktuális indexét, mert ez a mező a régebbi értéket tartalmazza, mielőtt meg a kattintás megtörtént. Valójában az OnChange metódus meg arra is használható, hogy tiltsa az oldalváltást az AllowChange paraméter beállításával, ami alapbeállításként False.

Alternatívaként kijelölhetőek az oldalak a nevük használatával (a jegyzettömb ActivePage tulajdonságával) az indexük használata helyett (a PageIndex tulajdonság). Ez csak akkor működik, ha az oldalak nevei megegyeznek a fülekével:

```
procedure TForm1.TabSet1Change(Sender: TObject;
  NewTab: Integer; var AllowChange: Boolean);
begin
  Notebook1.ActivePage := TabSet1.Tabs [NewTab];
end;
```

Hogy a programot érdekessé tegyük, néhany komponenst hozzá kell adnunk a különféle oldalakhoz. Tervezéskor, a jegyzettömb oldalain dolgozva meg kell változtatnunk ezek PageIndex tulajdonságainak értékeit; ezt nem tehetjük meg pusztán csak a fülekre kattintással mint a PageControl komponensnél. Amint beírunk egy új értéket a PageIndexben vagy az ActivePage tulajdonságban az Object Inspectorban, a jegyzettömb látható oldala eszerint változik. Egy jobb alternatíva, ha a Form Designerben (Form Tervező) kiválasztjuk a jegyzettömböt, és bekapcsoljuk a SpeedMenu-t(GyorsMenü), aminek vannak parancsai a következő illetve előző oldalra lapozáshoz.

Most, hogy tudjuk, hogyan kell megváltoztatni tervezéskor az aktív oldalt, mi van a példa oldalain? Úgy gondoltam, hogy az oldalakhoz hozzáadok pár vezérlőt, amiket ezen két komponens tulajdonságainak megváltoztatására használhatunk. Láthatjuk ezen két komponens leírását a Tab1 példa formjának következő szöveges leírásában:

```
object Form1: TForm1
  Caption = 'Tab Test App'
  OnCreate = FormCreate
  object Notebook1: TNotebook
    Align = alClient
    object TPage
      Caption = 'One'
      object Label1: TLabel...
      object ListBox1: TListBox...
    end
```

```
    object TPage
      Caption = 'Two'
      object Label2: TLabel...
      object GroupBox1: TGroupBox
        Caption = 'Dithering'
        object RadioButton1: TRadioButton
          Caption = 'On'
          OnClick = RadioButton1Click
        end
        object RadioButton2: TRadioButton
          Caption = 'Off'
          Checked = True
          OnClick = RadioButton2Click
        end
      end
    end
    object TPage
      Caption = 'Three'
      object Label3: TLabel...
      object ColorGrid1: TColorGrid...
    end
    object TPage
      Caption = 'Four'
      object Label4: TLabel...
      object Memo1: TMemo...
      object ChangeButton: TButton...
    end
    object TPage
      Caption = 'Five'
      object Label5: TLabel...
      object Label6: TLabel
        Caption = 'Hello !'
        Font.Color = clRed
        Font.Height = -96
      end
    end
  end
  object TabSet1: TTabSet
    Align = alBottom
    DitherBackground = False
    Tabs.Strings = (
      'Pages'
      'Dithering'
      'Colors'
      'Tabs'
      'Hello')
    OnChange = TabSet1Change
  end
end
```

Vajon, mit csinál az első oldal? Tartalmaz egy listapanelt az oldalak neveivel, és kitölti a jegyzettömb oldalainak neveivel, amikor a form létrejön. Ezúttal a kód egyszerűbb, mert mindkét komponens a TStringList objektumon alapszik:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListBox1.Items := Notebook1.Pages;
  Memo1.Lines := TabSet1.Tabs;
  {reset the page}
  Notebook1.PageIndex := 0;
end;
```

Szintén egyszerűsíthetjük a kódot az oldalak lapozására, amikor a felhasználó a listapanel egyik elemére kattint. Amint megváltoztatjuk a jegyzettömb aktív oldalát, csak arra kell vigyáznunk, hogy kiválasszuk a megfelelő fület is, mivel az oldal és fül indexek szinkronizációja nem automatikus:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  Notebook1.PageIndex := Listbox1.ItemIndex;
  TabSet1.TabIndex := Listbox1.ItemIndex;
end;
```

A második oldalnak két rádiógombja van, hogy be- vagy kikapcsolhassuk a TabSet vezérlő hátterének dithering hatását.

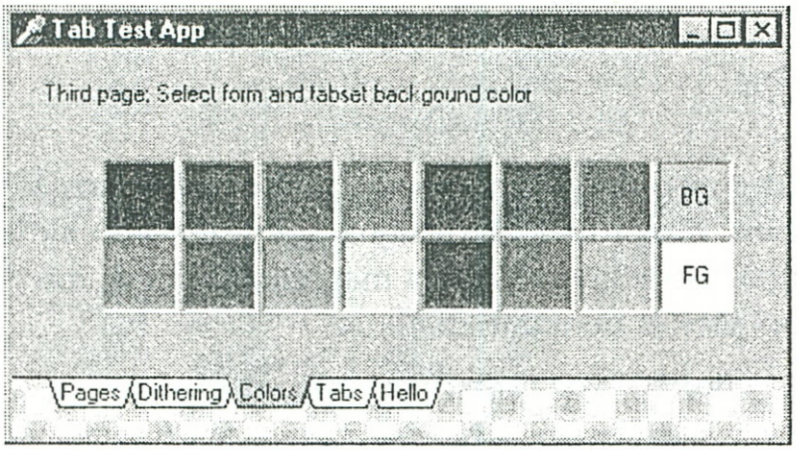
A harmadik oldal tartalmazza a ColorGrid komponenst, amit a form háttér színének beállításához használhatunk, és amit a jegyzettömbhöz is használunk. Ezért jó ötlet lehet ennek használata a fülbeállítások és az aktív fül hátterének a beállításához is. A fontos ötlet az, hogy a megfelelő háromdimenziós hatás eléréséhez, az aktív fül színe mindig meg kell egyezzen a jegyzettömb színével, ezért úgy fog tűnni, mint a felszínének egy meghosszabbítása (lásd 14.8. ábra). A második szín, az előtér színe, a nem aktív fülekre használandó:

```
procedure TForm1.ColorGrid1Change(Sender: TObject);
begin
  Form1.Color := ColorGrid1.BackgroundColor;
  TabSet1.BackGroundColor:=ColorGrid1.BackgroundColor;
  TabSet1.SelectedColor:=ColorGrid1.BackgroundColor;
  TabSet1.UnselectedColor:=ColorGrid1.ForegroundColor;
end;
```

A negyedik oldal lehetővé teszi a felhasználónak hogy megváltoztassa a fülek szövegét (de nem a Memo oldalakét). Tartalmaz egy Memo mezőt, ami a FormCreate metódusbeli szokásos beállításokat tartalmazza (lásd az alábbi kódot). A Change gomb kódja:

```
procedure TForm1.ChangeButtonClick(Sender: TObject);
begin
  if Memo1.Lines.Count <> TabSet1.Tabs.Count then
```

```
        MessageDlg (
          Format('Not %d lines!', [TabSet1.Tabs.Count])',
          mtError, [mbOk], 0)
    else
      TabSet1.Tabs := Memo1.Lines;
end;
```



14.8. ábra *A jegyzettömbnek beállítható új színek és a dithering hatás példája, valamint a Tab1 példa fülei*

Ha az oldalaknak hosszabb, még kifejezőbb neveket adunk, a TabSet komponens létre tud hozni két mini görgetőgombot a fülek mentén görgetéshez. Ez, és az előbb megbeszélt Pages példa két érdekes szempontra világít rá:

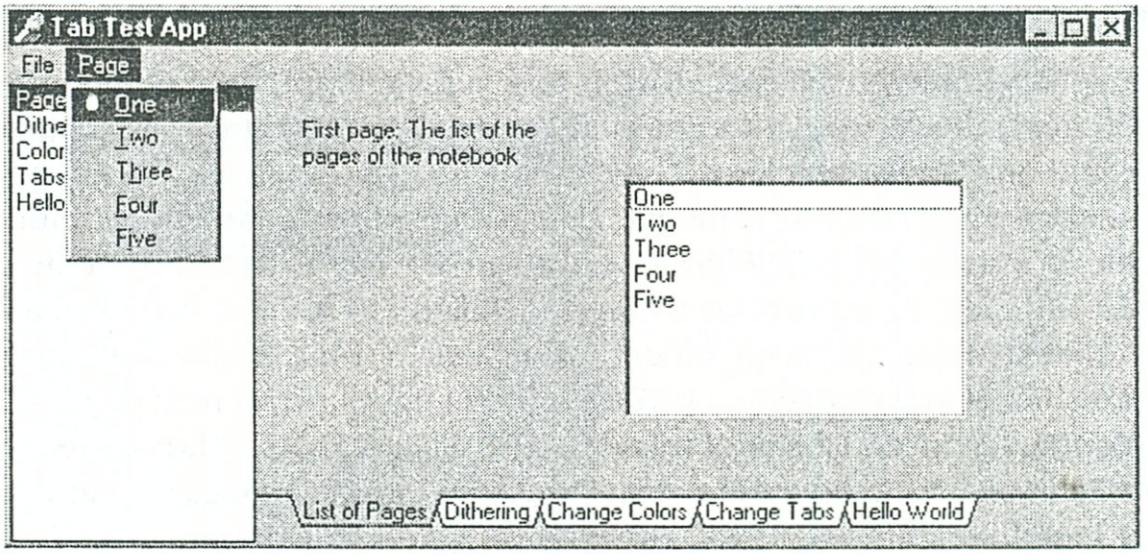
- Nincsen megszorítás a fülek használatára egy PageControl vagy egy jegyzettömb lapjának változtatásánál. Az első oldalon levő listapanel, amit erre a célra használtunk, elhelyezhető az egyik oldalra és oldalválasztóként használható, amint a következő fejezetben tesszük.
- Igen egyszerű a fülek feliratainak szerkesztése. Ez a technika használható arra, hogy megengedjük egy felhasználónak az alkalmazás konfigurálását.

Jegyzettömb oldalának megváltoztatása

Az imént befejezett példa rengeteg irányban továbbfejleszthető, különösen újabb, alternatív módot nyújtva egy jegyzettömb oldalának változtatásához. Ugyanez mondható el a Pages példáról. Mindemellett, amikor a standard Windows 95 komponenst használjuk, nem olyan fontos más módszereket vizsgálgatni az oldal megváltoztatásához, mint akkor, amikor egy kevésbé szabványos felhasználói interfészt használunk. Ha egy teljesen egyedi felhasználói felületet készítünk, egy jegyzettömb könnyen használható fülek nélkül.

Az első, általunk használt tehnika szerint egyszerűen adjunk hozzá egy, az első oldal jegyzettömbéhez hasonló listapanelt, és tegyük láthatóvá az ablakban. Ekkor csak annyi a teendő, hogy rákattintunk erre a listapanelre, és így másik oldalra váltunk. A második hozzáadandó komponens egy menü lesz, amellyel kiválaszthatjuk a használni kívánt oldalt.

A Tab2 példa elkészítéséhez, egy új listapanelt hozhatunk létre a form bal, vagy jobb oldalán. Hogy ezt elérjük, a különféle komponenseket át kell méretezzük, ki kell kapcsolnunk azok felülre igazítását és keretként egy fix keretet kell választanunk, amit így nem lehet átméretezni. A 14.9. ábrán látható a végső form egy példája. A program elindításakor kitölthetjük az új listapanelt ugyanarra a szövegre, mint a TabSet komponens címkéi.



14.9. ábra *A Tab2 program mutatja a kijelölt oldalt mind a fülben, mind pedig a baloldali listapanelben, és a Page legördülő menüben egy rádiógomb-szerű jellel*

Az új listapanel elemén történő kattintást kezelni tudjuk (ListBox2) azzal, hogy írtunk egy, az első listapaneléhez hasonló kódot (ha csak átmásoljuk és beillesztjük ezt a kódot, akkor ne felejtsük el a listapanelek számait is átírni). Sokkal jobb, ha megírjuk a kódnak egy olyan verzióját, ami mindkét listapanelt kezeli, és összekapcsoljuk mindkettejük kattintási eseményével:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  {works with both list boxes}
  Notebook1.PageIndex := (Sender as TListBox).ItemIndex;
  TabSet1.TabIndex := (Sender as TListBox).ItemIndex;
end;
```

Ezzel a kóddal, amikor a listapanelen mást jelölünk ki, egy új oldalt választunk ki, és kijelöljük a megfelelő fület. A Tab-2ben bemutatott másik technika az, hogy létrehozunk egy legördülő menüt a különféle oldalak kiválasztásához. Minden egyes menühöz rendelt OnClick esemény kapcsolatban van egy önálló metódussal, a MenuPage-dzsel, ami egy egyszerű trükkön alapszik: minden egyes menü neve megfelel a jegyzetömb oldalainak nevének. Ez egy kicsit egyszerűsíti a kódot:

```
procedure TForm1.MenuPage(Sender: TObject);
begin
  Notebook1.ActivePage := (Sender as TMenuItem).Name;
  TabSet1.TabIndex := Notebook1.PageIndex;
  ListBox2.ItemIndex := Notebook1.PageIndex;
  CheckCurrentPage;
end;
```

Ez a megközelítés könnyen fejleszthetővé is teszi a programot, mert nem szükséges új kód írása, illetve megváltoztatása akkor, ha új oldalakat és a hozzájuk tartozó menük mezőjét hozzuk létre. Csak a fenti első sor tartalmazza a menüadatok kiválasztásához tartozó eseményt. A másik két sor arra kell, hogy felülírjuk a fülek és a listapanel kijelöléseit. Az utolsó sor egy CheckCurrentPage nevű eljárást hív meg. Ez a metódus a menü mezőjén egy jelölést helyez el az aktuálisan kijelölt oldalnak megfelelően, amint már a 14.9. ábrán mutattuk. Ezen metódusnak el kell tüntetnie a jelölést minden egyes menü mezőjén (beállítja minden egyes mező Checked tulajdonságát False-ra), és ekkor engedélyezi az aktuális oldalhoz tartozóét, esetleg használ egy case elágazást, ami a Notebook1.PageIndexre épül. Ez az, ahogyan ezt a példát készítettük.

Ahhoz, hogy egy jobb CheckCurrentPages eljárást írjunk és fejlesszük a felhasználói felületet, rádiómenüket kell használnunk a Pages legördülő menühöz, és ki kell törölni azt a fenti öt sort, ami minden egyes lehetséges mező kijelölését megszüntette. Valójában az elágazást is el kell tűntetni, mert az aktív oldalak száma megegyezik a kijelölendő menü mezőinek számával. A metódus végső verziója:

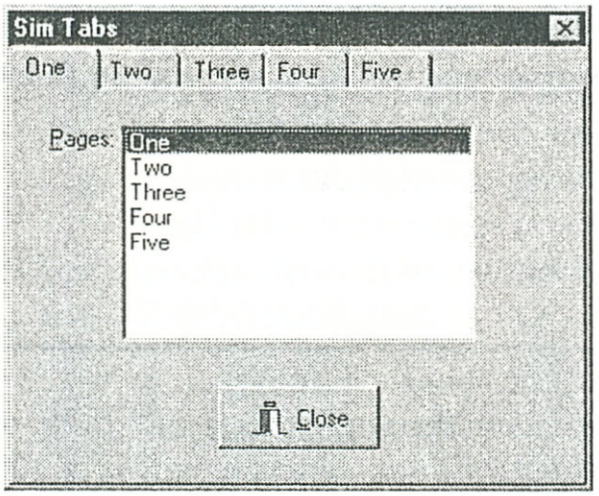
```
procedure TForm1.CheckCurrentPage;
begin
  Page1.Items [NoteBook1.PageIndex].Checked := True;
end;
```

Természetesen, szinte sosem lesz szükség három különböző módszerre azért, hogy kiválasszuk egy jegyzettömb oldalát, amint a Tab2 példában mutattuk, de egy pár lehetőség hasznos lehet, hogy a különféle felhasználók igényeit kielégítse. Ezen példa célja az volt, hogy megmutasson néhány, a PageControl komponenshez is alkalmazható lehetőséget (különösen a legördülő menüt használó oldalkiválasztást).

Füllel rendelkező jegyzettömbök

Egy harmadik lehetőség a PageControl, TabSet és Notebook komponens használatára a TabbedNotebook komponens használata. Ezen komponens felhasználói felülete igen hasonlít a Windows 95 PageControljához, de ez szintén elérhető volt a Delphi 16 bites változatában, vagyis ez Windows 3.1-es alkalmazásokban is használható.

A SimTab nevű példa, amit a TabbedNotebook használatához készítettünk, hasonlít, bár valamivel egyszerűbb, mint a korábban bemutatott Page és Tab1 példák. Amint a 14.10. ábrán látható, ezen többboldalas form első oldalának van egy, az oldalak neveivel ellátott listapanelje. Van egy Close (Bezár) gomb is, ami a jegyzettömb minden oldalán szükséges. Újra felhívom a figyelmet: a gombot a formra raktuk, nem a jegyzettömbre, és utána vittük át a jegyzettömbre.



14.10. ábra *A SimTab példa első oldala*

A gombot alul középen helyeztük el, és kiválasztottunk a formnak egy nem méretezhető keretet. A SimTab példa jegyzettömbjének többi oldala megengedi, hogy a fülek területének fontjait, és soronként a fülek számát megváltoztassuk. A két utolsó oldal szinte teljesen üres.

E példa kódja egészen egyszerű. Valójában a TabbedNotebook komponens automatikusan kezeli az oldalváltást úgy, mint a PageControl. Ez azt jelenti, hogy nem szükséges az aktív oldalt minden egyes alkalommal, amikor a felhasználó egy fülre kattint, megváltoztatnunk, mint a Tab1 és Tab2 példákban. Itt ismét több mint egy technikát alkalmazhatunk az oldalak megváltoztatásához. Ahogy korábban is, kitöltetjük az első oldal listapaneljét az oldalak neveivel, és átvihetjük másik oldalra, ha a felhasználó a listapanel megfelelő elemére kattint.

A füles jegyzettömb oldalának megváltoztatásához beállíthatjuk az oldal nevét vagy indexét, mint a többi többboldalas komponensét. Miután a listapanelnek egy nulla-alapú ItemIndex tulajdonsága van, ami a kijelölt mezőre utal, egyszerűen megírhatjuk a következő kódot:

```
TabbedNotebook1.PageIndex := ListBoxPages.ItemIndex;
```

A jegyzettömb másik két oldala használható a fontok, és a fülek sorrendjének megváltoztatásához. A megfelelő gombra kattintással a betűtípus beállítása egyszerűen történhet a FontDialog komponens használatával, írjuk be a kívánt értéket a fülek fontjának megváltoztatásához:

```
if FontDialog1.Execute then
  TabbedNotebook1.TabFont := FontDialog1.Font;
```

A program egy tesztcímke Caption-jét is megváltoztatja, hogy jelezze az aktuális font nevét, és egy példa arra, hogyan fog kinézni a font. Az utolsó művelethez, a fülek soronkénti számának megváltoztatásához egszerűen létrehoztunk egy SpinEdit komponenst és egy gombot. A kód egyszerű. Amikor a felhasználó rákattint a gombra, a SpinEdit komponens értéke a soronkénti fülek számaként lesz használva:

```
TabbedNotebook1.TabsPerRow := SpinEdit1.Value;
```

Figyelem!

Sajnos ez a kód nem a dokumentációnak megfelelően működik, a TabsPerRow tulajdonság megváltoztatásának nincsen hatása a fülekre! Úgy döntöttem, meghagyom a rossz kódot azért, hátha a Borland kijavítja a hibát, még ha ez egy régi tulajdonság is, amit a Delphi 1-gyel való kompatibilitás miatt hagytak bent.

Jegyzettömb fülek nélkül és fülek jegyzettömb nélkül

A PageControlon alapuló felület kezd eléggé általánossá válni, és a NoteBook és TabSet komponens természetesen illeszkedik egymáshoz. Mindemellett a két komponens egyidejű használata nem az egyetlen lehetőség. Amint a fejezet címe sugallja, írhatunk alkalmazásokat fülekkel többboldalas komponensek nélkül, vagy jegyzettömbökkel, egyetlen fül- vagy listafajta nélkül. Íme néhány példa, némelyikük a következő fejezetekben teljességgel felhasználásra kerül.

A TabControl (vagy TabSet) önmagában többféleképpen felhasználható: egy többoldalas szerkesztőben (mint például a Delphi szerkesztője), egy többoldalas bitmap-nézegetőben, hogy egy fájlválasztó ablakban meghajtót válasszunk, egy háromdimenziós táblázatkezelőben, vagy hogy kijelölhessünk egy formot egy többformos programban. A Delphi füleket használ az oldalak kijelölésére a komponenspalettában is.

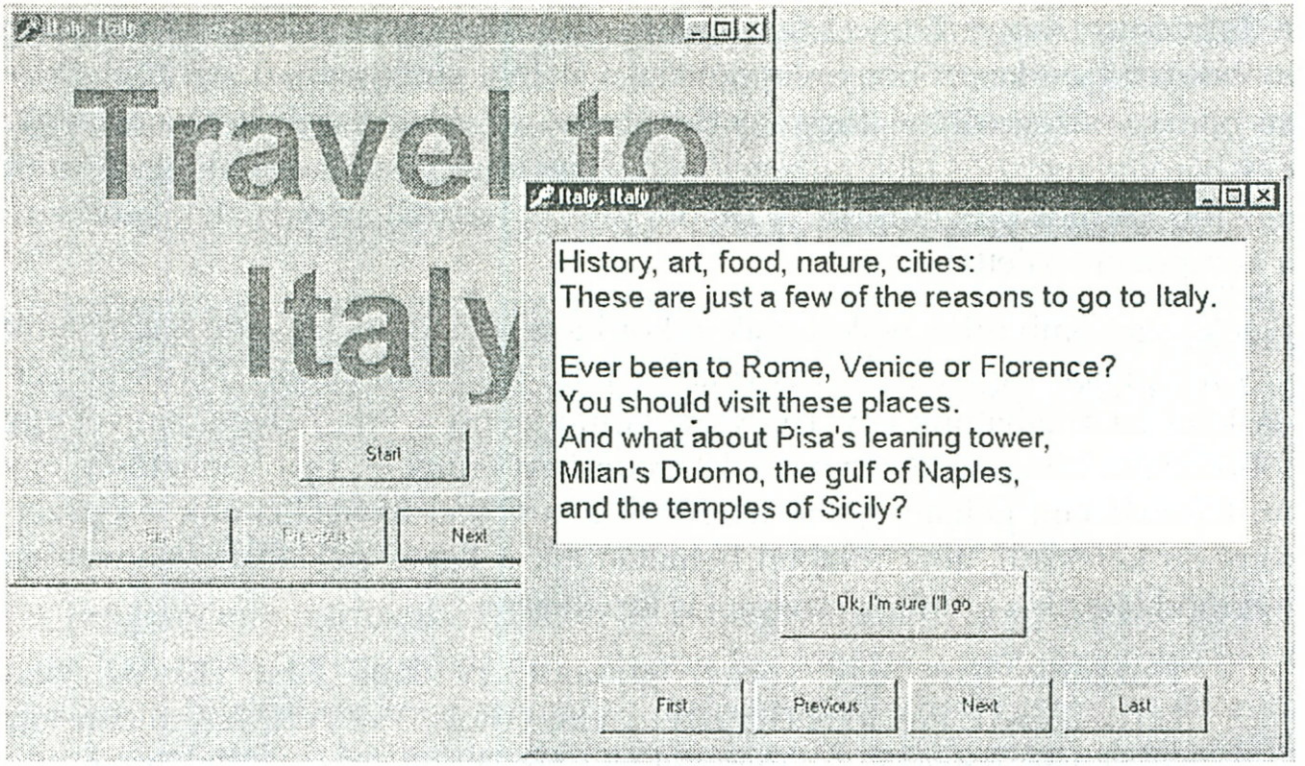
Egy jegyzettömb fülek nélkül csak akkor használható, ha helyette használunk egy listapanelt vagy egy menüt (mint a Tab2 példában láttuk). Másik megoldásként használhatunk egy TreeView komponenst a lista helyett, amivel egy többoldalas böngészőjéhez hasonló eredményt érünk el. Újabb lehetőség egy fül használatára néhágy gomb létrehozása, amikkel navigálhatunk a jegyzettömbön keresztül, amit részben bemutattunk a Pages példában. Alapvetően két mód létezik a gombok navigációs kezelésére:

- Hozzunk létre egy sornyi gombot, például First (Első), Next (Következő), Previous (Előző) és Last (Utolsó) névvel, hasonlóan egy adatbázis navigálási parancsaihoz. Ennek egyik speciális esete egy Expert vagy varázsló fejlesztése. Ezekben az alkalmazásokban a form egy jegyzettömbre épül, és mint általában van egy gombja a következő oldalra lépéshez, és egy másik az előzőre visszalépéshez. Tipikusan csak akkor lehetséges a következő oldalra lépni, ha már állítottunk pár opciót. Az oldalak sorrendje (és még a megjelenésük is) az előző beállításoktól függhet.
- A jegyzettömb különböző oldalaihoz rendeljünk speciális gombokat, hogy más, kapcsolódó oldalakra mehessünk. Ez a megoldás hasonlít a hiperhivatozások használatához, és egy kép, vagy egy szöveg kijelölt részét használhatjuk gombok megnyomása nélkül.

Megjelenítés egy jegyzettömbben

Első példánk egy fül nélküli jegyzettömb, neve NoteOnly. Van az alján egy gombsor, illetve pár gomb az oldalakon a különleges navigációs parancsoknak. A form alján lévő négy gomb egy panelra került, így a jegyzettömb el tudja foglalni a form kliens területének a maradékát. Talán a legjobb út a példa megértéséhez a futtatása. A program egy Olaszországban tett utazást mutat be.

Minden egyes oldalnak van egy gombja a következő lépésre váltáshoz, amint a 14.11. ábrán látható, de esetenként két vagy több választás adódik. Semmi más nincs a gombok mögött, és a kódjuk is igen egyszerű. Általában, amikor rákattintunk egy gombra, a jegyzettömb egy új oldalát választjuk ki. Némivel több kódolást igényel kikapcsolni a jelenleg jelentés nélküli gombokat. Például, miután rákattintunk a First gombra, ki kell kapcsolni az első két gombot (mivel a felhasználó már az első oldalon van és ott nincsen előző oldal) és engedélyezni kell a másik két gombot abban az esetben ha tiltva voltak.



14.11. ábra *A NoteOnly példa oldalai, egy kis olaszországi bemutatóval*

Kicsivel összetettebb a Next és Previous gombok esete. Ellenőrizni kell, hogy a jegyzettömb elérte-e az első vagy az utolsó oldalt, és ennek megfelelően tiltani vagy engedélyezni kell a megfelelő gombokat. Az igazi probléma itt az, hogy szintén engedélyezni és tiltani kell a gombok némelyikét a gombsorban, amikor a jegyzettömb különféle oldalainak navigációs gombjaira kattintunk, hogy elkerüljük az ellentmondásos viselkedéseket.

Egy képkezelő program

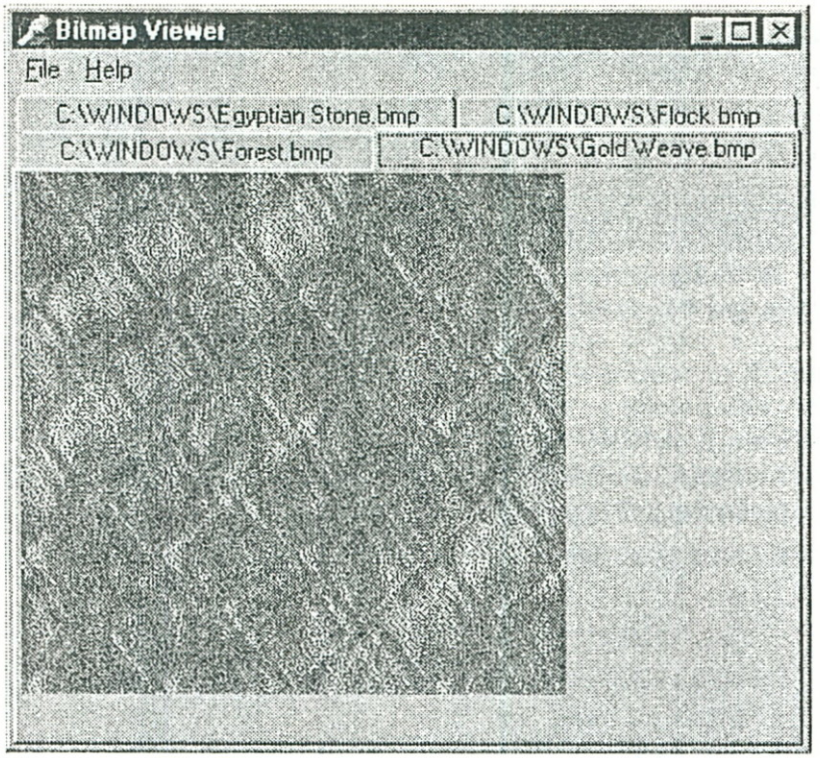
Most, hogy végigmentünk egy fülek nélküli jegyzettömb példáján, készen állunk megvizsgálni a másik esetet: egy fület, ami nem kapcsolódik jegyzettömbhöz. Gyakran találhatunk a fülhöz kapcsolódó más komponenseket, például egy panelhez kapcsolhatunk egy fülkészletet, egy képet, a sokféle rács egyikét, vagy más típusú komponenseket. A következő TabOnly nevű példánkban meg szeretnénk jeleníteni egy bitmapet egy Image(kép) komponensben. A megjelenő kép függ a fülek fenti kiválasztásától (amint a 14.12. ábrán látható). Ez egy egyszerű példa. A formnak van egy TabControl komponense a teljes kliens területhez rendelve, és egy Image komponense a TabControlon belül, és lefedi a saját kliens területét. A másik két komponens áll egy főmenüből és egy fájl megnyitó párbeszédablakból (File Open dialog box), ami megengedi a többszörös kijelölést:

```
object Form1: TForm1
  Caption = 'Bitmap Viewer'
  object TabControl1: TTabControl
    Align = alClient
    Tabs.Strings = ('None')
    OnChange = TabControl1Change
    object Image1: TImage
      Align = alClient
    end
  end
  object OpenDialog1: TOpenDialog
    Options = [ofAllowMultiSelect, ofPathMustExist,
      ofFileMustExist, ofShareAware]
    Title = 'Select one or more images'
  end
  object MainMenu1: TMainMenu
    object File1: TMenuItem...
      object Open1: TMenuItem...
      object N1: TMenuItem...
      object Exit1: TMenuItem...
    object Help1: TMenuItem...
      object AboutImageViewer1: TMenuItem...
  end
end
```

Kezdetben a TabControlnak csak egy tiltot füle van, jelzi az állapotot (nincs fájl kijelölve). Amikor a felhasználó kiválasztja a **File Open**-t (fájlmegnyitás), a párbeszédablak megjelenik. A felhasználó kiválaszthat több fájlt, és a string tömbje a fájlok neveivel (az OpenDialog1 komponens Files tulajdonsága) a fülek feliratként felhasználásra kerül (a TabControl1 Tabs tulajdonsága):

```
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    TabControl1.Tabs := OpenDialog1.Files;
    TabControl1.TabIndex := 0;
    TabControl1Change (TabControl1);
  end;
end;
```

Miután kiraktuk az új füleket, frissítenünk kell a őket, hogy az első fülnek megfelelő képet mutasson.Ehhez a program meghívja a TabControl Onchange eseményéhez kapcsolt metódust, ami betölti az aktuális fülhöz tartozó fájlt:



14.12. ábra *A képnézegető interfész a TabOnly példában*

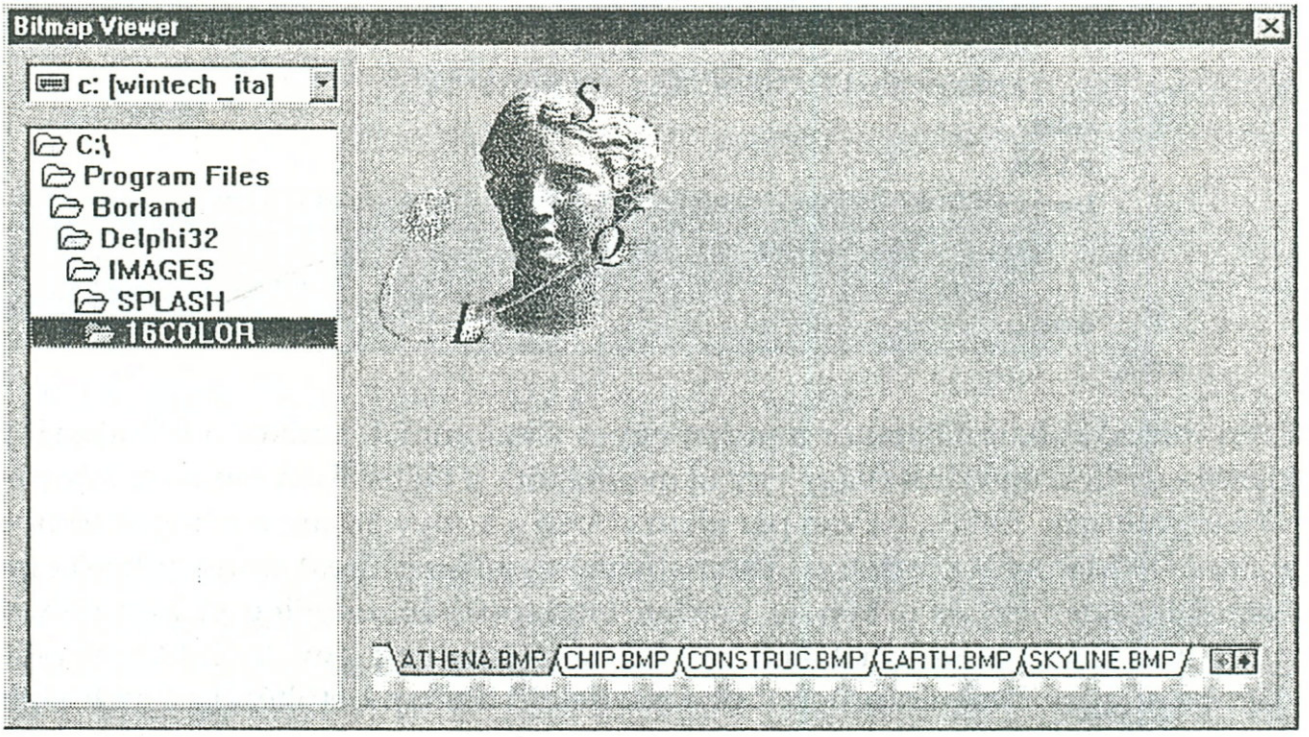
```
procedure TForm1.TabControl1Change(Sender: TObject);
begin
  Image1.Picture.LoadFromFile (
    TabControl1.Tabs [TabControl1.TabIndex]);
end;
```

Ez a példa működik, de nem túl jól. Az egyik probléma, hogy a teljes elérési út kijelzése a fület nagyon naggyá teszi, és elfoglalja a képernyő java részét (ez azért van így, mert a TabControl Multiline tulajdonságát True-ra állítottuk). Hogy ezt a problémát megoldjuk, a program definiál egypár rejtett, átmeneti stringet, így az útvonalnév kivehető a fülből. Másik probléma az, hogy ha kiválasztunk egy fájlt, ami nem bitmapet tartalmaz, hibaüzenetet kapunk. Mindemellett a program figyelmeztetni fogja a felhasználót egy szabványos üzenettel, figyelmen kívül hagyja a fájlt, és folytatja a program futását.

Az utolsó probléma a használhatósággal kapcsolatos. Ez a program nem igazán használható a merevlemezen lévő képek keresésére, mert az OpenDialog futtatása újra és újra szükséges. Jobb ötlet egy olyan könyvtárlista alkalmazása közvetlenül a formon, amin keresztül a felhasználó kereshet. Ezt a Delphiben egyszerű megcsinálni, amióta van egy készen elérhető DirectoryListBox (könyvtár listapanel) komponens. Pár fejlett rendszerkomponens és néhány programsor használatával írhatunk egy teljesen kiforrott képnézegetőt.

Képnézegető fülekkel

Legújabb TabOnly2 nevű képnézegető példánban rakjunk egy DriveComboBox-ot és egy DirectoryListBox-ot egy új formba, ahogy azt a 14.13. ábra mutatja. A form jobb oldalára helyezzünk egy TabControlt egy Image komponenssel a belsejében. Még egy Bevel komponenst is adjunk hozzá a formhoz, hogy tökéletesítsük a felhasználói felületet.



14.13. ábra *A képnézegető a TabOnly2 példában*

A programban rejlő ötlet az, hogy ha a felhasználó új könyvtárat választ ki, a füleknek azonnal ki kell jeleznie a könyvtárban levő bitmap fájlok nevét. Először, hozzá kell rendelnünk a kombinált listát a meghajtók neveihez és a könyvtárlista panelhez, így minden egyes változtatás a kombinált listában megjelenik a többi komponensben. Ezt egyszerűen megtehetjük a DriveComboBox komponens DirList tulajdonságának állításával. A második lépés a bitmap fájlok neveinek betöltése a fülekbe minden egyes alkalommal, amikor a kijelölés megváltozik a könyvtárlistapanelben. Megtehetjük ezt kézzel is, DOS függvények használatával, vagy más komponenst használhatunk a probléma megoldására. A második megoldás illik a Delphi filozófiájához és sokkal egyszerűbb, ezért ezt a megoldást követjük.

Rakjunk a formra egy FileListBox komponenst, és csökkentsük a méretét úgy, hogy láthatatlan legyen, majd helyezzük a másik komponens mögé, vagy egyszerűen állítsuk a Visible tulajdonságát False-ra. Ezután állítsuk át a fájllista

Mask tulajdonságának értékét *.bmp-re. Most adjuk a könyvtárlistapanel OnChange eseményéhez egy metódust, és írjuk be a következő kódot:

```
procedure TForm1.DirectoryListBox1Change(Sender:
    TObject);
begin
  with FileListBox1 do
    if Items.Count = 0 then
    begin
      TabControl1.Tabs.Clear;
      Image1.Visible := False;
      TabControl1.Tabs.Add ('None');
    end
    else
    begin
      Image1.Visible := True;
      TabControl1.Tabs := FileListBox1.Items;
    end;
end;
```

Ezen metódus használatakor minden egyes alkalommal, amikor a felhasználó kijelöl egy új könyvtárat (vagy egy új meghajtót), a fájllistában mutatott könyvtár megváltozik. Ekkor, ha van pár elem ebben a könyvtárban, a stringek tömbje átmásolódik a fül készletbe. Alternatívaként, a None(üres) string adható egy üres fülkészlethez, és a kép az eredeti bitmappal átmenetileg el lesz rejtve. Amikor a fülek kijelzik a fájlok nevét, használhatunk egy, a TabOnly példáéhoz nagyon hasonló programot a megfelelő bitmap betöltéséhez az Image komponens képén:

```
procedure TForm1.TabControl1Change(Sender: TObject);
begin
  with TabControl1 do
    if Tabs [TabIndex] <> 'None' then
      Image1.Picture.LoadFromFile (Tabs [TabIndex]);
end;
```

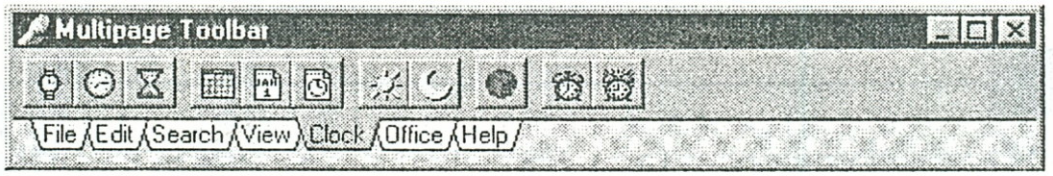
Egy többoldalas eszköztár

Utolsó példánk, mielőtt továbmennénk a form elválasztásokhoz, definiál egy jegyzettömböt fülekkel, amiket a többoldalas eszköztár létrehozásához használunk. Egy többoldalas eszköztár létrehozása általában könnyű. Csak néhány gombot kell elhelyezni egy PageControl vagy egy jegyzettömb oldalain. Egy ilyen eszköztárat láthatunk a 14.14. ábrán vagy ismerkedjünk ezzel a MultiBar példa futtatásával, amely egy TabSheet komponensen és egy jegyzettömbön alapszik. Elhatároztuk, hogy nem írjuk be a TabSet komponens füleinek feliratát, hanem egyszerűen futás közben bemásoljuk az oldalak nevét a jegyzettömb füleibe:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TabSet1.Tabs := Notebook1.Pages;
end;
```

Természetesen, a fülek helyes működésének érdekében, mint mindig, most is hozzá kell rendelni őket a jegyzettömbhöz:

```
procedure TForm1.TabSet1Change(Sender: TObject;
  NewTab: Integer; var AllowChange: Boolean);
begin
  Notebook1.PageIndex := NewTab;
end;
```



14.14. ábra *A MultiBar példa kimenete futáskor*

Vegyük észre, hogy a példa fülei utalnak a lehetséges legördülő menük neveire. Jobb megoldás lenne két vagy három eszköztár használata megfelelően a különféle környezeti helyzeteknek (fájl szerkesztés, mappa kezelés, más eszközök, és így tovább). A program automatikusan megválthatoztathatja az eszköztár oldalát, ha a tartalom változik, de a felhasználó továbbra is szabadon választhat ki egy másik eszköztár oldalt. Egy többboldalas eszköztárnál, mint például a Delphi komponenspalettája, jobb alternatíva lehet sok eszköztár megléte helyett a teljes képernyőterület kihasználása, amint sok alkalmazás is teszi.

Tipp

Kölcsönvettem ezen fejezet példáihoz az összes bitmapet a Delphi Image Library-ból, e képgyűjtemény általában elérhető a `Delphi3\Images` könyvtárból és annak alkönyvtáraiból.

Egyszerű példa készítésével az a probléma, hogy minden gombnak kézzel kell kiválasztani a bitmapet, ami eléggé unalmas. A megoldás, hogy nevet adunk a gomboknak, így aztán később be tudják tölteni az azonos nevű bitmapet futás közben. Ha ezt a megközelítést választjuk, újabb fejlődést érhetünk el a

bitmapek forrásonkénti tárolásával a futtatható fájlon belül, a külső BMP fájlok használata helyett. A másik probléma a programmal, hogy rengeteg Windows erőforrást köt le. Ennek megoldása a képek oldalankénti dinamikus betöltése lehet, kikapcsolva a gombokat, amikor az oldal rejtve van.

Formfelosztási technikák

Rengeteg út kínálkozik a form elválasztási technikáinak kivitelezéséhez a Delphiben, és a Delphi 3-ban még közvetlen támogatást is kapott egy speciális komponenssel, a Splitterrel (elválasztó), ami a komponenspaletta Additional (további) oldalán található. Mégis, szeretnénk bemutatni pár hasznos módszert, beleértve a közvetlen egérkezelést.

Tipp

Az új TCoolBar komponens (a 11. fejezetben láttuk) szintén használható elválasztóként, habár nem ez az elsődleges célja.

Lényegében a többször használt példát fogjuk elkészíteni, a hivatalos módszerrel kezdve, azután megnézzük a HeaderControl (fejléckezelő) komponens használatát, és végül az alacsony szintű egérkezelést tárgyaljuk. A minden egyes alkalommal bemutatott példa egy három, állatok neveit tartalmazó listapanellel ellátott formon alapul. Ezek a listapanelek kétszeres kattintással a font változtatást szolgálják.

Elválasztás Splitterrel

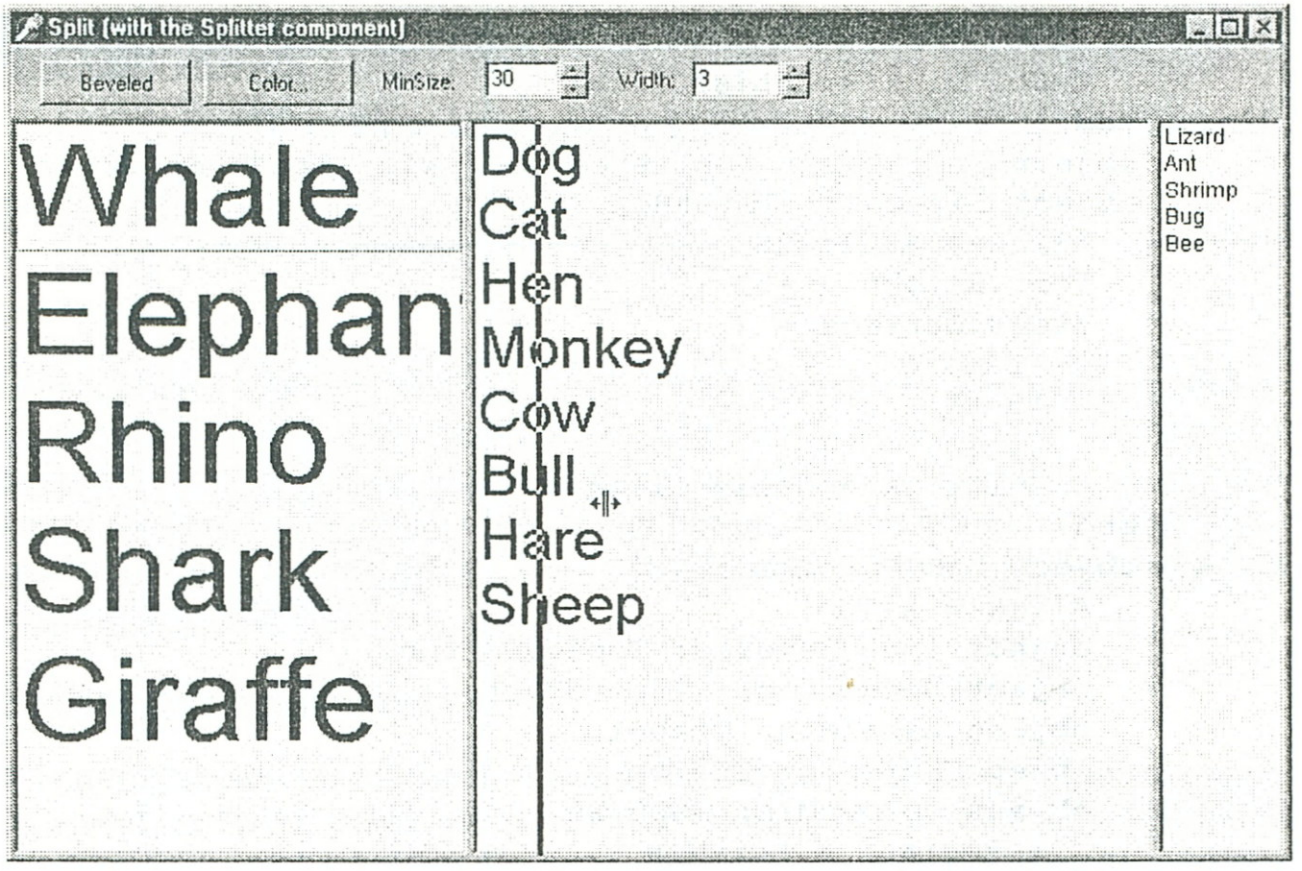
A legtermészetesebb módszerrel kezdjük a dolgot, a Splitter komponens használatával. A Split1 példa elkészítéséhez rakjunk egy formba egy Listapanel komponenst majd a Splitter komponenst, egy második Listapanelt, még egy Splittert, és egy utolsó Listapanel komponenst. A formnak egy egyszerű eszköztára két SpeedButton komponenssel és kétféle címkével, szerkesztőpanele, és UpDown vezérlője is van. A form tartalmaz egy FontDialog és egy ColorDialog komponenst is, amint a szöveges leírásból jól látható:

```
object Form1: TForm1
  Caption = 'Split (with the Splitter component)'
  object ListBox1: TListBox
    Align = alLeft
    Font.Height = -64
    Items.Strings = (
      'Whale'
```

```
      'Elephant'
      ...)
    OnDblClick = OnListClick
  end
  object Splitter1: TSplitter
    // Align = alLeft {default, not listed}
    Cursor = crHSplit
    Beveled = False
  end
  object ListBox2: TListBox
    Align = alLeft
    Font.Height = -32
    Items.Strings = (
      'Dog'
      'Cat'
      ...)
    OnDblClick = OnListClick
  end
  object Splitter2: TSplitter...// similar to Splitter1
  object ListBox3: TListBox
    Align = alClient
    Font.Height = -12
    Items.Strings = (
      'Lizard'
      'Ant'
      ...)
    OnDblClick = OnListClick
  end
  object Panel1: TPanel
    Align = alTop
    object SpeedBeveled: TSpeedButton...
    object SpeedColor: TSpeedButton...
    object LabelMin: TLabel...
    object EditMin: TEdit...
    object UpDownMin: TUpDown
      Associate = EditMin
      Min = 10
      Max = 200
      Increment = 10
      Position = 30
      OnClick = UpDownMinClick
    end
    object LabelWidth: TLabel...
    object EditWidth: TEdit...
    object UpDownWidth: TUpDown.. //similar to UpDownMin
  end
  object ColorDialog1: TColorDialog...
  object FontDialog1: TFontDialog...
end
```

Egyszerűen elhelyezve a két elválasztó komponenst, a formot a teljes futás közbeni mozgatási és vezérlő átméretezési lehetőséggel látjuk el. A Splitter komponens Width (szélesség), Beveled (ferde), és Color (szín) tulajdonságok a külalakját befolyásolják, és a Split1 példában az eszköztár használható azok megváltoztatására. Újabb idevonatkozó tulajdonság a MinSize (minimum méret), ami a form különféle komponenseinek a minimális méretét adja meg. A szétválasztási művelet alatt (14.15. ábra) egy vonal jelzi az elválasztó végleges helyét, de ez csak bizonyos határokon belül mozgatható.

Azt javaslom, próbáljuk ki a példát, így tökéletesen megértjük az elválasztó hatását szomszédos, illetve a form többi vezérlőjére. A példa kódja olyan egyszerű, hogy nem is írjuk le. A részletek a mellékelt CD-n találhatóak.



14.15. ábra *Az elválasztó komponens meghatározza a form minden egyes vezérlőjének minimális méretét még akkor is, ha azok nem szomszédosak magával az elválasztóval*

Vízszintes elválasztás

A Splitter komponens vízszintes elválasztásra is használható, az alapértelmezésbeli függőleges elválasztás helyett. Mindazonáltal ez a módszer egy kicsit komplikáltabb. Lényegében egy formra kell helyezni egy komponenst, felülre igazítani, és az elválasztót a formra helyezni. Alapbeállítás szerint ekkor az balra igazított lesz. Válasszuk ki az Align tulajdonság alTop értékét, majd kézzel méretezzük át a komponenst, megváltoztatva a Height tulajdonságot az Object Inspectorban (vagy a komponens átméretezésével).

A SplitH példában egy vízszintes elválasztóval ellátott form látható. Ennek a programnak két jegyzetkomponense van, amikkel egy fájlt nyithatunk meg, és egy elválasztó található közöttük. Szintén tartalmaz egy státuszsort, ami a jegyzet komponensek aktuális magasságát követi nyomon. A szöveges leírás:

```
object Form1: TForm1
  Caption = 'Horizontal Splitter'
  OnResize = Splitter1Moved
  object MemoUp: TMemo
    Align = alTop
    ScrollBars = ssVertical
    OnDblClick = MemoDblClick
  end
  object Splitter1: TSplitter
    Cursor = crVSplit
    Align = alTop
    OnMoved = Splitter1Moved
  end
  object MemoDown: TMemo
    Align = alClient
    ScrollBars = ssVertical
    OnDblClick = MemoDblClick
  end
  object StatusBar1: TStatusBar...
  object OpenDialog1: TOpenDialog
end
```

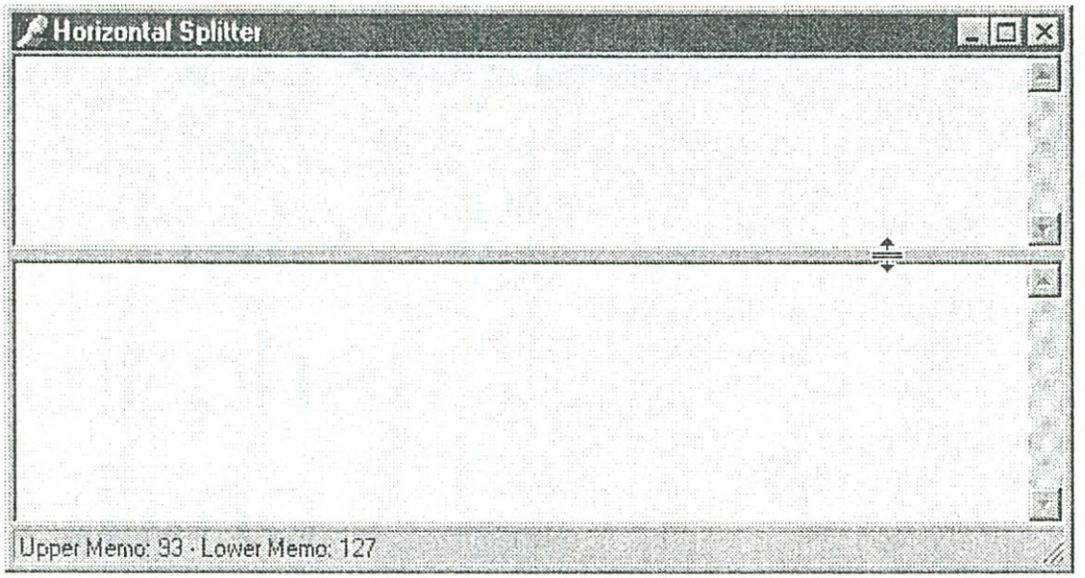
Amikor egy jegyzeten kétszer kattintunk, a program beletölt egy szöveges fájlt (figyeljük a komponens struktúráját):

```
procedure TForm1.MemoDblClick(Sender: TObject);
begin
  with Sender as TMemo, OpenDialog1 do
    if Execute then
      Lines.LoadFromFile (FileName);
end;
```

A program az elválasztó OnMoved komponensét is kezeli (ez ennek a komponensnek az egyetlen eseménye), hogy frissítse a státuszsor feliratát. A form átméretezésével egyező kódot futtatunk:

```
procedure TForm1.Splitter1Moved(Sender: TObject);
begin
  StatusBar1.Panels[0].Text := Format (
    'Upper Memo: %d - Lower Memo: %d',
    [MemoUp.Height, MemoDown.Height]);
end;
```

A kód hatása látható a 14.16. ábrán, vagy a SplitH példa futtatásakor.



14.16. ábra *A SplitH komponens státuszsora mutatja a vízszintes elválasztó komponens pozícióját*

Alternatív elválasztó komponensek

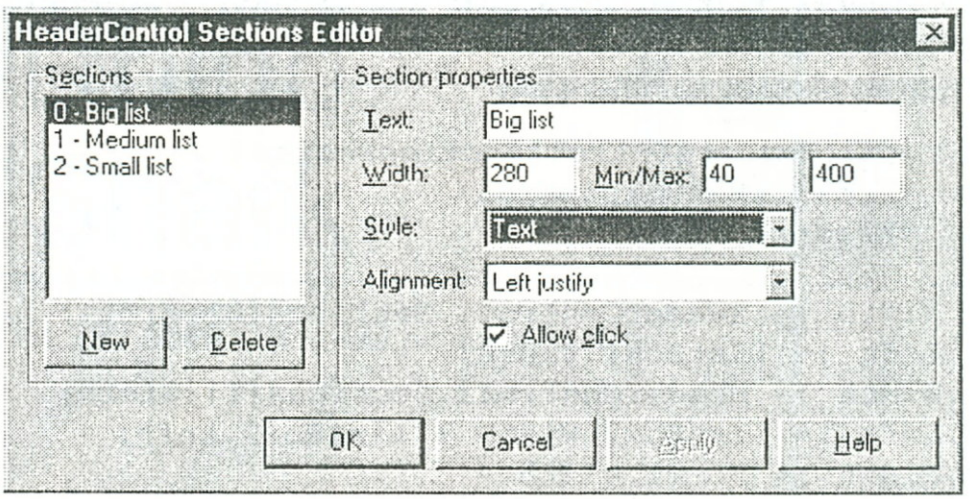
A Delphi 3 előző két, hivatalos elválasztási módszerének példája után nézzünk néhány alternatív megoldást. Habár ezen megoldásoknak több értelmük volt a Splitter komponens megjelenése előtt, lehetőséget adnak néhány új, elválasztáskor használható komponens bemutatására: a Header Control (fejléc vezérlő) és a Panels (panelek). Később egy messzebb mutató példát mutatok be, ami sokkal inkább a Windows egérüzenetein alapszik, mint Delphi komponenseken.

Elválasztás fejléccel

A HeaderControl használata egy újabb szokványos eljárás függőleges elválasztáskor. Alternatívaként (a Windows 3.1 kompatibilitás miatt) használható a Delphi saját Header komponense.

Ha egy formra HeaderControl komponenst rakunk, automatikusan felülre lesz igazítva. Ekkor hozzáadhatjuk a három listapanelt a form kliens területének a maradékához. Az első panel balra igazítható, de ezúttal ezt nem tehetjük meg a másodikkal és a harmadikkal. A prombléma az, hogy a fejléces terület kiszedhető a form látható felületéről. Ha a panelek automatikus igazításúak, akkor nem vehetők ki, amint a program ezt igényli is.

Most már definiálhatjuk a fejléc területét., a Sections tulajdonságok speciális szerkesztőjét használva. Amint a 14.17. ábrán látható, ez a tulajdonság szerkesztő a területek nevein túl különféle beállítások változtatását engedi meg. Beállítható a szöveg igazítása, az aktuális, minimális és maximális fejlécméret stb. A határértékek beállítása igen hatékony eszköz, mert lehetővé teszi, hogy elkerüljük ezt futáskor a kódon belül.



14.17. ábra *A HeadControl komponens Sections tulajdonságának szerkesztője*

Az egyetlen probléma a fejléc utolsó részével van. A harmadik listapanellel szeretnénk feltölteni a kliensterületet ha lehetséges, de a fejlécnek általában van egy kis üres része az utolsó rész után. Emiatt a Split2 példában a fejléc utolsó részét nagyon nagynak csináltam (lást 14.18. ábra). A HeaderControl tulajdonságainak értékei a form szöveges leírásából kivéve:

```
object HeaderControl1: THeaderControl
  Sections = <
  item
```

```
      MaxWidth = 400
      MinWidth = 40
      Text = 'Big list'
      Width = 280
    end
    item
      MaxWidth = 400
      MinWidth = 40
      Text = 'Medium list'
      Width = 180
    end
    item
      MaxWidth = 1000
      MinWidth = 40
      Text = 'Small list'
      Width = 1000
    end>
    OnSectionClick = HeaderControl1SectionClick
    OnSectionResize = HeaderControl1SectionResize
end
```

Amint a következő listából látható, egyszerre két eseményt kezelünk: OnSectionResize és OnSectionClick. Az elsőben egyszerűen újraméretezzük a három listapanelt a három résznek megfelelően:

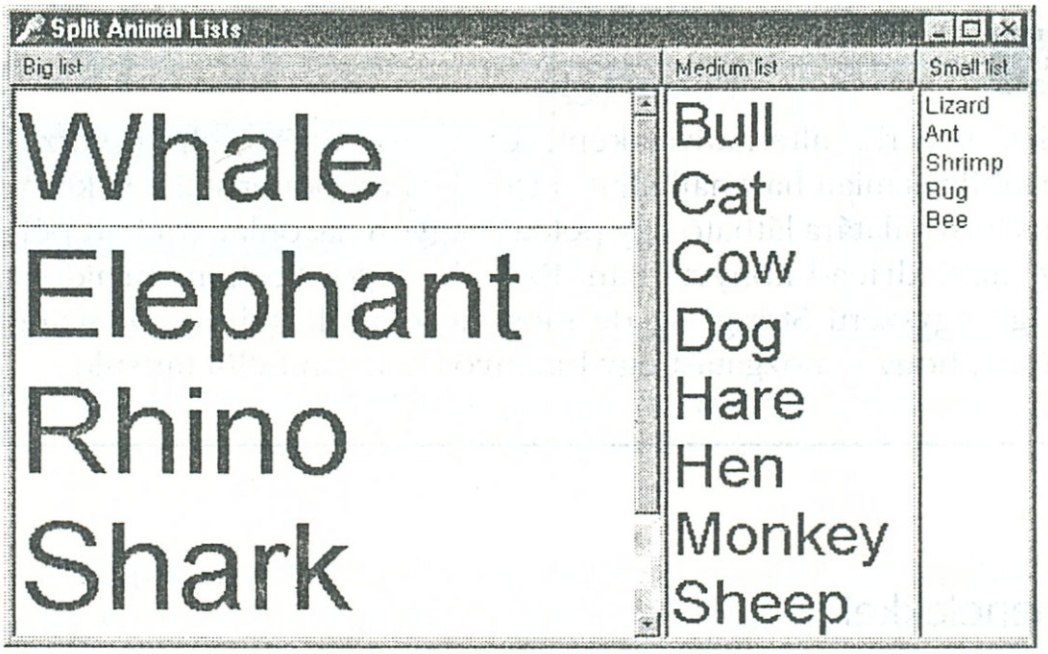
```
procedure TForm1.HeaderControl1SectionResize(
    HeaderControl: THeaderControl;
    Section:THeaderSection);
begin
  ListBox1.Width := HeaderControl1.Sections[0].Width;
  ListBox2.Left := ListBox1.Width;
  ListBox2.Width := HeaderControl1.Sections[1].Width;
  ListBox3.Left := ListBox2.Width + ListBox2.Left;
  ListBox3.Width := Form1.Width -
    HeaderControl1.Sections[0].Width -
    HeaderControl1.Sections[1].Width;
end;
```

Figyelem!

Form1.Width-et írtam a tulajdonság közvetlen elérése helyett, mert a THeaderControl komponens OnSectionTrack eseményének kezelőjének (akkor hívódik meg, amikor a felhasználó áthúzza az elválasztókat) van egy Width paramétere. Ez a paraméter elrejti a form vonatkozó tulajdonságát.

Ezen esemény alatt kezelnünk kell a form átméretezését a következő kód használatával (emlékezzünk arra, hogy az első listapanel automatikus igazítású):

```
procedure TForm1.FormResize(Sender: TObject);
begin
  ListBox2.Height := ListBox1.Height;
  ListBox3.Height := ListBox1.Height;
  HeaderControl1SectionResize(HeaderControl1,
    HeaderControl1.Sections [0]);
end;
```



14.18. ábra *A Split2 példa kimenete*

Miután beállítottuk a listapanelek magasságát, ez a metódus egyszerűen az előzőt hívja, olyan paramétereket átadva, amiket ezen példában nem fogunk használni. A HeaderControl második eljárása, a területre kattintás arra szolgál, hogy a megfelelő listapanel elemeit sorba rendezze. A probléma az, hogy honnan tudjuk, melyik részre kattintottak? A legjobb módszer, amit kitaláltam (biztos létezik ennél jobb is) az, hogy hasonlítsuk össze a paraméterként átadott területi objektumokat a HeaderControl Sections listatulajdonságának területi objektumaival:

```
procedure TForm1.HeaderControl1SectionClick(
  HeaderControl: THeaderControl; Section:
THeaderSection);
begin
  if Section = HeaderControl1.Sections[0] then
```

```
      ListBox1.Sorted := True;
    if Section = HeaderControl1.Sections[1] then
      ListBox2.Sorted := True;
    if Section = HeaderControl1.Sections[2] then
      ListBox3.Sorted := True;
end;
```

Természetesen ez a kód nem viselkedik a Windows alapbeállítása szerint az elemek sorbarendezésénél, vagyis amikor a fejlécre kattintunk, és fordított sorrendbe rendez újabb kattintásra. Ehhez meg kell írni egy saját rendező algoritmust.

Megjegyzés

A fejléc vezérlő alternatívájaként, és a 16 bites Delphi verzió kompatibilitás miatt használhatjuk a Header komponenst. Ezen komponens használatára látható egy példa (nagyon hasonlatos a leírt példához) az OldHead könyvtárban. Ezúttal a fejléc Sections tulajdonsága egy egyszerű StringList, de most kezelnünk kell az OnSizing eseményt, hogy a mozgatást egy bizonyos határon belül tartsuk.

Elválasztás panelekkel

A következő, Split3 példánkban az elválasztó a Panel komponensen alapszik a Splitter komponens helyett. Ezt a példát azért készítettem, hogy bemutassam a Delphi továbbfejlesztett kattint-és-húz lehetőségeit.

Ezen harmadik verzióhoz rakjunk egy formba három listapanelt és középre két vékony függőleges panelt (lásd 14.19. ábra). Most, a harmadik listapanelt kivéve, minden komponenst igazítsuk balra. Amint már bizonyára kitaláltuk, a harmadik listapanel lesz a kliens területhez rendelve. Vegyük észre, hogy be kell állítani a komponensek elrendezését a megfelelő sorrendben, az első listapaneltől kezdve, tovább az első panelt, majd a második listapanelt, a második panelt, és végül a harmadik listapanelt.

Ezúttal a problémát bármely Delphi komponens automatikus áthúzási tulajdonságának használatával kell megoldani. A példa két panele támogatja az automatikus áthúzási theirDragMode tulajdonságot, ami dmAutomatic-ra van állítva, és van bennük speciális vízszintes elválasztó kurzor (crHSplit) mind a standard, mind pedig a felhasználói kurzor számára. Mindhárom listapanel ugyanazt a kurzort használja áthúzáshoz.

14.19. ábra *A Split3 példa formja tervezéskor. Felhívom a figyelmet a panelekre a listapanelek között*

Mindegyik panel elfogadja a panel vagy panelek áthúzását a keretük közelében, és azonnal megfelelően méretezi a listapaneleket. Nem csinálhatunk 40 pixelnél keskenyebb listapanelt. Például csak az első panel húzható át az első listapanelen (Source=Panel1) és a kurzor vízszintes koordinátája (X) nem lehet 40 pixelnél kevesebb:

```
procedure TForm1.ListBox1DragOver(
  Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  if (Source = Panel1) and (X > 40) then
  begin
    Accept := True;
    ListBox1.Width := X;
  end
  else
    Accept := False;
end;
```

Az Accept-et megint True-ra kell állítani amikor egy panelt önmagán húzunk át, amikor a felhasználó lenyomja a bal egérgombot a panel felett de nem mozgatja onnan el azt. Ekkor kell megjeleníteni az áthúzási kurzort a normál helyett:

```
procedure TForm1.PanelDragOver(
  Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  if Source = Sender then
    Accept := True
  else
    Accept := False;
end;
```

Ezúttal a program ellenőrzi, hogy az eredetileg húzott komponens (Source) megegyezik-e azzal, amelyik megérkezik a húzási eseménykor (Sender). Ha ez a helyzet, akkor egy panelt önmagára húzunk, és nem egy másik panelen keresztül. Mivel megírtuk az eljárást ezzel az általános kóddal (speciális panelekre utalás nélkül), ezért mindkét panelhez használhatjuk.

Az első panel a második listapanelre is húzható. Ebben az esetben szintén újra kell méreteznünk, meg kell növelnünk az első listapanelt, de ellenőriznünk kell azt is, hogy a harmadik nem össze megy egy bizonyos értéknél kisebbre:

```
procedure TForm1.ListBox2DragOver(
  Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  if (Source = Panel1) and (ListBox3.Width > 40) then
  begin
    Accept := True;
    ListBox1.Width := ListBox1.Width + X;
  end
  ...
```

Most X érték relatív a második listapenelhez képest, így használhatjuk az első méretének növelésére. A maximális szélesség ellenőrzés, amit ez a kód használ messze van a tökéletestől, mivel ellenőrzi, hogy a harmadik listapenel elég nagy-e, és nem azt, hogy nagyobb lesz-e 40 pixelnél az aktuális egérmozgatási üzenet után. A kód le tudná ellenőrizni, hogy a harmadik listapanel jövőbeni mérete elég nagy lesz-e, de ebben a példában talán erre nincs szükség. Ha egy 30 pixel széles listapanelnél megállunk, ez nem nagy probléma. Egyszerűen kijavíthatjuk ezt azáltal, hogy megnöveljük a tesztelt értéket (például 45-öt használunk 40 helyett, megengedve 5 pixelt minden egyes egérmozgatás üzenetnek).

A ListBox2DragOver metódusnak ellenőriznie kell, hogy amikor a második panelt elkezdjük áthúzni a második listapanel felett, nem lesz-e a listapanel túlságosan kicsi. A harmadik listapanelnek szintén van egy OnDragOver eseménykezelője a második panelre vonatkozóan, ahogy az a mellékelt lemezen lévő végső kódban is látható.

Megjegyzés

Senki sem mondhatja, hogy az egér mozgatása két egymást követő egérmozgatási üzenet között történik. A rendszer nem küldhet üzenetet minden egyes pixelnyi mozgásnál, amíg az egeret igen lassan mozgatjuk. Az egérüzenetek diszkrétek, nem folytonosak. Generálásuk függ a rendszer időzítőitől és hasonló hardver-orientált paraméterektől, de hozzáadódnak egy üzenet-sorhoz. Igazából egy alkalmazás adott sebességgel kap egérmozgatási üzeneteket, ami függ a rendszer teljes terheltségétől. A lépésenkénti 5 pixel egy kitalált szám.

Elválasztó vonal rajzolása

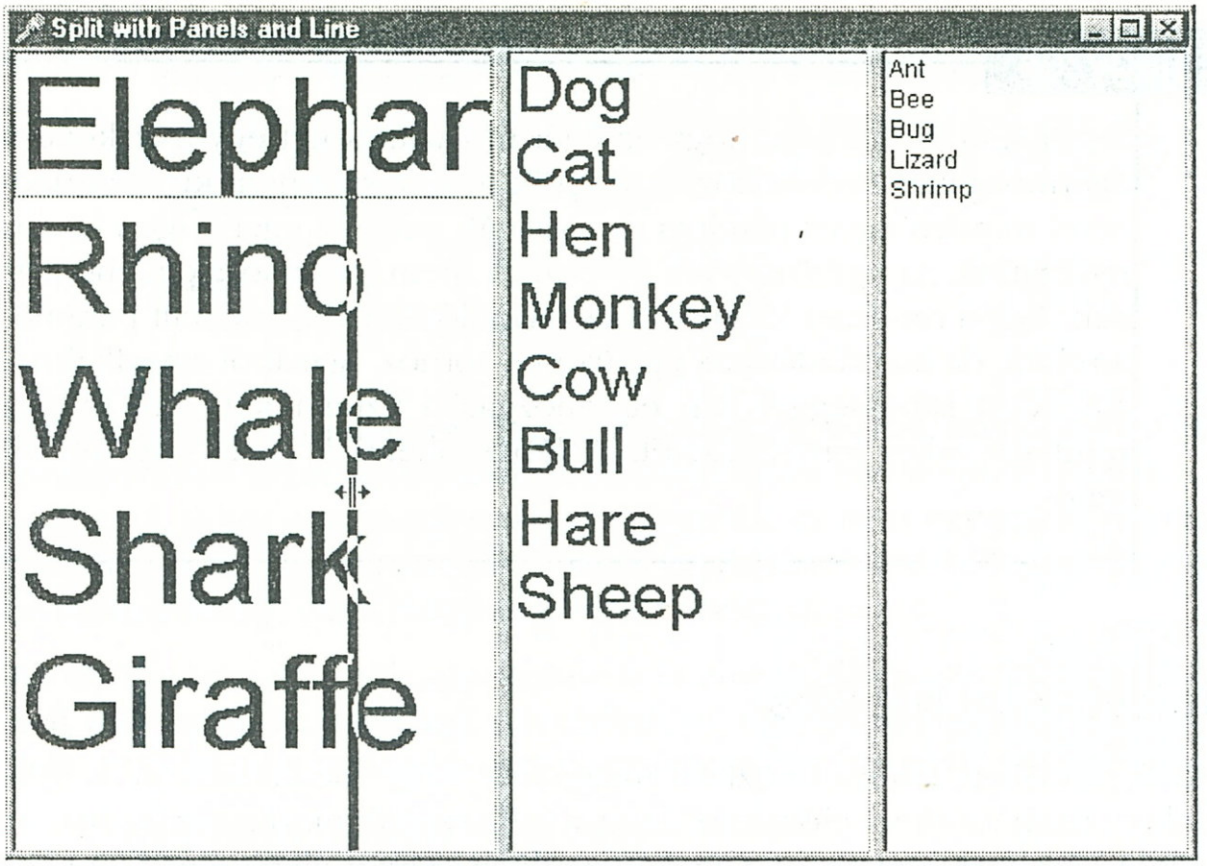
Az előző példában áthúzás közben mozgattuk vagy méreteztük át a listapaneleket . Ezen módszer előnye az, hogy a felhasználó mindig tiszán látja, mi történik. A hátránya, hogy a listapanelek tartalmát számos alkalommal újra kell rajzolni, gyakran csúnyán villog, és ha túl gyorsan méretezzük át a listapaneleket, a program késni fog az ablakok újrarajzolásával, ami nagy szürke területeket fog eredményezni.

A Split1 példában, ami egy igazi elválasztó komponensen alapult, áthúztunk egy nagy függőleges vonalat, jelezve az elválasztó lehetséges végső pozícióját anélkül, hogy mozgattuk volna a komponenst amíg az áthúzás véget nem ért. Ez az a módszer, amit a következő, Split4 című példában használtam.

Az elválasztó elkészítésekor rajzoljunk egy vékony szürke vonalat a listapanelre az aktuális vízszintes pozícióba. A probléma az, hogy ki kell törölnünk a szürke vonalat, amint az elválasztó elmozdul. Ezért a programnak emlékeznie kell, hol volt a vonal, illetve képesnek kell lennie eltárolnia a listapanel előző tartalmát (annak szövegét). Megoldás egy vonal rajzolása a pmNot toll mód használatával, ami megváltoztatja az aktuális kimenetet. Ez hasznos, mert a kimenet kétszeri megváltoztatásával egyszerűen eltárolhatjuk az eredeti rajzolatot. Hogy ezt a hatást elérjük minden egyes listapanelnél, írjuk be az alábbi inicializáló kódot:

```
ListBox1.Canvas.Pen.Width := 5;
ListBox1.Canvas.Pen.Mode := pmNot;
```

A Split4 formja, amint a 14.20. ábra mutatja, ugyanolyan komponenssel van ellátva, mint az előző verzió, de egy komplexebb forráskóddal.



14.20. ábra *A Split4 példa vonalelválasztója*

Vegyük észre, hogy két egyedi metódust írtam. A DragDrawLine metódus arra használandó, hogy rajzoljon meg egy új vonalat és eseményenként törölje a régebbit. A másik metódus, DeleteDragLine arra való, hogy törölje az aktuális vonalat, amikor az áthúzás befejeződik. A vonal megrajzolásához vagy törléséhez ezek a függvények az aktuális, vagy az előző listapanel felületének MoveTo és LineTo metódusát használják (paraméterként átadva). A két metódus kódja:

```
procedure TForm1.DrawDragLine (
  List: TListBox; X: Integer; State: TDragState);
begin
  {if there is currently a line...}
  if OldList <> nil then
  begin
    {delete the previous line, redrawing onto it}
    OldList.Canvas.MoveTo (OldX, 0);
    OldList.Canvas.LineTo (OldX, ClientHeight);
  end;

  if not (State = dsDragLeave) then
  begin
    {if the user is not leaving the list box,
```

```
      draw the new line and store the older values}
      List.Canvas.MoveTo (X, 0);
      List.Canvas.LineTo (X, ClientHeight);
      OldX := X;
      OldList := List;
    end
    else
    begin
      {if the user is leaving the list box, then do
      *not* draw and set old values to nil/invalid}
      OldX := -10;
      OldList := nil;
    end;
  end;

  procedure TForm1.DeleteDragLine (List: TListBox);
  begin
    {delete the old line and reset the invalid position}
    if OldList <> nil then
    begin
      OldList.Canvas.MoveTo (OldX, 0);
      OldList.Canvas.LineTo (OldX, ClientHeight);
      OldX := -10;
      OldList := nil;
    end;
  end;
```

Vegyük észre, hogy a vonalrajzolási eljárás teszteli az áthúzási esemény által átadott State paraméter értékét, ellenőrzésként a dsDragLine-nal. Ha az egérkurzor elhagyja a listapanelt, a vonal nem rajzolódik újra. Ellenkező esetben a form OldList és OldX privát mezői nulla és egy negatív véletlen számértékre állítódik be külön-külön. A Split4 példa másik fontos része annak tesztelése, hogy az OldList objektum jelenleg nullát, vagy egy listapanelt tartalmaz-e. Ez a teszt bármely objektum metódusának hívása előtt megtörténik. Ha ezt nem tennénk, megkockáztatnánk egy nem definiált objektumhoz való hozzáférést, ami kifogásolható.

A három OnDragOver esemény kódjában a Split3 példában látotthoz hasonlót figyelhetünk meg. Megnézzük, hogy a listapanel elfogadhat-e áthúzást az aktuális panelből (kivéve a ListBox2-t, ami mindkettőből fogadhat), és akkor megnézzük, hogy az aktuális pozíció az engedélyezett területen belül van-e. Amikor az elválasztó értékhatáron belül van, a program meghívja a vonalrajzolás (vagy még jobb, a mozgatás) metódust. Amikor az elválasztó kiér a megfelelő területről, a program kitörli az utolsó sort,ez a DeleteDragLine.

Mint általában, a pozíció helyességének ellenőrzéséhez azt figyeljük, mikor lesz az X érték 40-nél kevesebb. Ha ez nem fordul elő, akkor az utolsó

listapanel számára elég hely van. Hogy ezt az ellenőrzést végrehajthassuk mindemellett szükséges kiszámolnunk a harmadik listapanel méretét, miután ez nem változik folyamatosan az áthúzási műveletek alatt, mint az előző példákban. A legegyszerűbb módszer ennek kiszámítása az, hogy amikor a harmadik listapanel elég nagy már ahoz, hogy megnézzük, a szélességi növekedés (X) mikor fogja elhagyni azt több, mint 40 pixelnél, amint a mellékelt lemezen a teljes listából látható. Egy egyszerűbb eset:

```
procedure TForm1.ListBox1DragOver(
  Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept := False;
  if Source = Panel1 then
    if X > 40 then
    begin
      Accept := True;
      DrawDragLine (ListBox1, X, State);
    end
    else
      DeleteDragLine (ListBox1);
end;
```

Az áthúzás alatt álló listapanel új szélessége csak akkor kerül beállításra, ha az áthúzási folyamat véget ér, mint a következő kódban:

```
procedure TForm1.ListBox1DragDrop(
  Sender, Source: TObject; X, Y: Integer);
begin
  DeleteDragLine (ListBox1);
  ListBox1.Width := X;
end;
```

Amint látható, nincsen semmiféle ellenőrzés, mivel az elengedés csak akkor jöhet létre, ha az áthúzás megengedhető. Mindemellett a második listapanel OnDragDrop metódusában szükséges megfontolnunk, melyik panelről származik az áthúzás.

Habár e példának összetettebb a kódja, és némi időbe került a teljes séma átgondolása, a hatás megérte az erőfeszítést. Különös kihívás volt a két különböző listapanelen lévő két egymást követő áthúzás esetének a kezelése (ami csak egy gyors egérmozgatás után kerül elő). E szituációt kezeléséhez mutattam be az OldList objektum ötletét.

Közvetlen egérkezelés a formban

Ötödik, és egyben az utolsó elválasztásos példánk kevésbé Delphi-orientált, inkább a Windows felé mutat. A listapanelen kívül más komponenst nem használ. Elválasztók, panelek vagy fejlécek, és a Delphi által más komponenseknek felajánlott egár áthúzási támogatás nélkül (formnak nem lehet áthúzási művelete a VCL-ben), a formnak fogadnia és kezelnie kell a közvetlen Windows egér üzeneteket. Ezen megközelítés másik hibája az, hogy hagynunk kell némi helyet a listapanelek között, hogy a felhasználó dolgozhasson a formon, és így elvesztjük az automatikus igazítás lehetőségét. Másrészt viszont precízen kezeljük vezérlőnek, annak ellenére, hogy nem használtunk a formon elválasztót, mint a Split4 példában.

Megjegyzés

Még egyszer, ezen példa célja nem az, hogy többféle utat mutasson az elválasztók létrehozására, hanem hogy szemléltesse, hogyan kell kezelni az egér áthúzásokat kizárólag a Windows API-val.

Vagyis itt van egy form három listapanellel. Nem látunk túl sokat a formból, csak két kis függőleges területet, közöttük az elválasztóként használandó listapanelekkel. A form a crHSplit és a standard kurzort használja. Amikor a felhasználó megnyomja a bal egérgombot a formon (azaz, a két vékony függőleges terület egyikén), a program elindít egy áthúzási kódot, ami addig folytatódik, amíg az egérgombot el nem engedi. Ezalatt az idő alatt, bármilyen egérmozgatás hatása a listapanelek mozgatásának hatásával egyezik meg, szimulálva a megfelelő mozgatást az elválasztási területen. Mielőtt megbeszéljük a példa kódját, tekintsünk ki egy picit arra, hogyan implementálja a Delphi az áthúzást és fogadja az egér inputját.

Húzás egérrel

Amint a 10. fejezetben megbeszéltük, a Windowsnak 3 csoportra osztható egér üzenete van: olyanok,l amik a gomb megnyomására, elengedésére vonatkoznak, illetve olyanok, amik az egérmozgatásra. A Delphi komponensek a bal gombos kattintás fogalmát is tartalmazzák, de nem tartalmazzák az áthúzást. Hogyan tudnánk létrehozni, és hogyan hozta létre a Borland a Delphiben ? Az áthúzás özlete igen egyszerű. A program kap egy sort gomb-le, egérmozgatás és gomb-fel üzenetekből. Amikor lenyomjuk a gombot, elkezdődik az áthúzás, habár a valódi esemény csak akkor következik be, amikor a felhasználó mozgatja az egeret (a gomb elengedése nélkül) és amikor az áthúzás megszakad (amikor a gomb-fel üzenet megérkezik).

A probléma ezzel az alapvető megközelítéssel az, hogy nem megbízható. Egy ablakot általában csak akkor ér egéresemény, amikor az egér a kliens területe felett van; vagyis amikor a felhasználó megnyomja az egérgombot, áthúzza az egeret másik ablakhoz, elengedi a gombot, a második ablak kapja meg a gomb-fel üzenetet.

Erre a problémára két megoldás van. Egyik (ritkán használt) az egér kivágás. A Windows egyik API függvényét hívva (ClipCursor), rákényszeríthetjük, hogy az egér ne hagyhassa el a képernyő bizonyos területét. Amikor a kijelölt területen kívülre húzzuk az egeret, az egy láthatatlan korlátba ütközik. A másik (általánosabb) megoldás az egéreseményekelfogása (capture). Amikor egy ablak magához ragadja az egeret, minden ezt követő egér input ennek az ablaknak továbbítódik, amint a 11. fejezet Fire példájában láttuk. Ez az a megközelítés, amit a Split5 példában használni fogunk.

A húzás kódja

A Split5 kódja három metódus köré épül: FormMouseDown, FormMouseMove, és FormMouseUp. A formon lenyomva a bal egérgombot (azaz, az elválasztásra használt vékony függőleges területek egyikén) elkezdődik a program, beállítja a form két logikai mezőjét, a Dragging-et és a FirstSplit-et. Az első változó arra jó, hogy megmutassa, az áthúzás folyamatban van-e, és ezt a változót fogja használni a másik két metódus. A második változó azt mutatja, a két elválasztó közül melyiket húzza jelenleg a felhasználó (True mutatja az elsőt, False a másodikat). A kód:

```
procedure TForm1.FormMouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
    Dragging := True;
    SetCapture (Handle);
    if X <= ListBox2.Left then
      FirstSplit := True
    else
      FirstSplit := False;
  end;
end;
```

Ezen metódus fontos eseménye a SetCapture API függvény hívása. Amikor az áthúzás történik, amit az ide vonatkozó változó (Dragging) jelez, és a felhasználó mozgatja az egeret, a program számos dolgot csinál, de az ötlet egyszerűen a listapanelek átméretezése, amíg a határukra nem érünk:

```
procedure TForm1.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  if Dragging then
    if FirstSplit then
    begin
      if (X > 40) and
        (X < ClientWidth - ListBox3.Width - 40) then
      begin
        ListBox1.Width := X - 2;
        ListBox2.Left := X + 2;
        ListBox2.Width := ListBox3.Left -
          ListBox2.Left - 4;
      end;
    end
    else
      {similar code for the second split}
      ...
```

Vegyük észre, hogy ezúttal az X a form koordinátáiban értendő, nem pedig annak a listapanelnek a koordinátája, ami felett az elválasztót húzzuk éppen át. Ha kiérünk a területről, a listapanel helyesen lesz mozgatva és rendezve. Ebben a példában, amikor az első elválasztót húzzuk, az első listapanel mérete megnő illetve a másodiké lecsökken a megfelelő mérték szerint. Emlékezzünk rá, hogy megint szükséges azt a kis helyet (4 pixelt) szabadon hagyni a listapanelek között, hogy az elválasztás végrehajtódjon.

Ha kimegyünk a területről, az áthúzási művelet ReleaseCapture API függvény meghívásával megszakad, és a program beállítja a Dragging mező értékét hamisra. Ugyanez történik, ha elengedjük az egérgombot:

```
procedure TForm1.FormMouseUp(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Dragging then
  begin
    ReleaseCapture;
    Dragging := False;
  end;
end;
```

Hogyan tovább?

A fejezet első részében láttuk, hogyan kezeli a Delphi a form görgetését, és hogyan tudunk dolgozni vele. Láttuk, hogyan rajzoljunk egy görgetődő felületre, és hogyan követhesse nyomon a görgetési műveleteket a felhasználó. A második részben olyan más technikákat vizsgáltunk, amiket használva

megnövelhetjük az egy formra kirakható komponensek számát, többboldalas komponensek különféle formjait használva. Láttuk, hogyan használjuk az új Windows 95-ös PageControl és TabCOntrol komponenseket, és hogyan kapcsolhatunk egy TabSet komponenst egy Jegyzettömb komponenshez. A harmadik részben megbeszéltük az új Splitter komponensre épülő elválasztási technikákat, a HeaderControl és panelek használatával. Szintén megvitattuk a Windows egérrel kapcsolatos üzeneteinek közvetlen kezelését. A Delphi komponensek igen rugalmasak. Ahelyett, hogy egy adott gondolkodásmódot kellene követnünk, rengeteg elérhető lehetőség közül választhatunk.

A jegyzettömb metódus kompolex alkalmazások létrehozását teszi lehetővé, sok formmal (vagy oldallal) és egy egyszerű felhasználói interfésszel, ami általában sokkal jobb, mintha rengeteg ablakot mozgatnánk a képernyőn. Másik módszer többformos, összetett alkalmazások készítésére az MDI technika. Ezt a következő, 15. fejezetben ismertetjük.

15. FEJEZET

MDI alkalmazások készítése

E fejezet tartalma:

- MDI a Windowsban: technikai áttekintés
- Keret és gyermekablakok a delphiben
- MDI alkalmazások különböző gyermekablakokkal
- Az MDI gyors elsajátítása

MDI alkalmazások készítése

Idáig megtanultuk, hogyan kezeljük az egyszerű, egyetlen főformmal rendelkező Delphi alkalmazásokat, valamint a bonyolultabbakat, melyeknek több különböző formjuk és párbeszédablakuk van. Láthattuk azt is, hogy a görgetés, szétválasztás és rétegezés (ahogy azt a Notebook használatakor láttuk) olyan technikák, melyekkel számos információt és összetevőt jeleníthetünk meg egyetlen formon belül.

A párbeszédablakok vagy a másodlagos formok használatán, valamint az összetevők egy formba való tömörítésén túl van egy harmadik módszer, ami igen elterjedt a Windows alkalmazásokban: az MDI (Multiple Document Interface, azaz Többdokumentumos Interfész) használata. Az MDI alkalmazások olyan formokból állnak, melyek egyetlen fő formon belül jelennek meg. Ezt a fejezetet néhány, a Windowson belüli MDI fejlesztéssel kapcsolatos általános és technikai tudnivalóval kezdjük, majd lépésről lépésre felépítünk egy MDI programot Delphiben.

MDI a Windowsban: technikai áttekintés

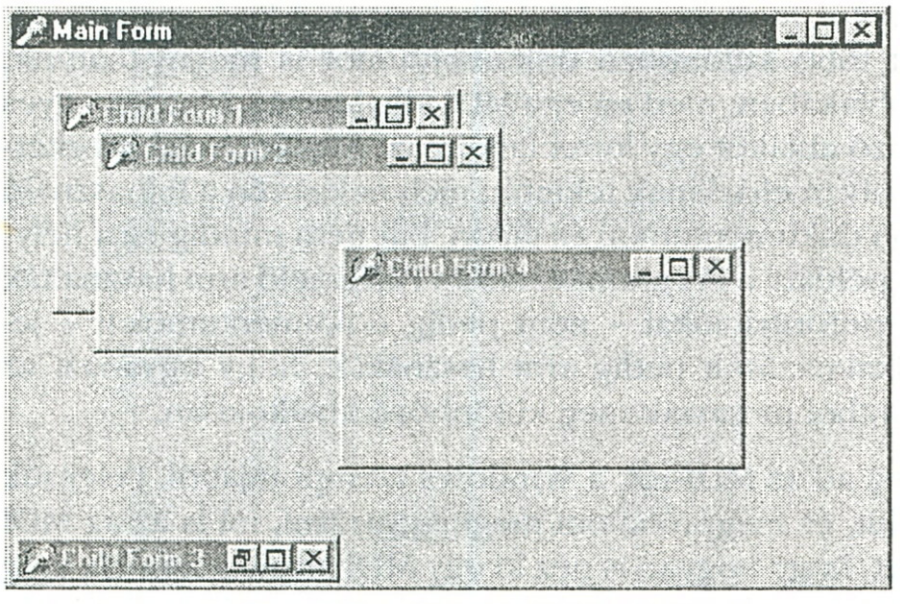
A Windows alkalmazásának kezdetén minden program a képernyőn lévő különböző ablakokból állt (ez a környezet második változatánál volt így, mivel a Microsoft operációs rendszerének első verziója csak egymás mellé tudta helyezni, de átfedni nem tudta őket). Néhány évvel később a Microsoft bevezetett egy olyan technikát, mellyel létrehozhatunk egy teljes értékű ablakot (nevezhetjük formnak) egy másik ablakon belül, melynek keret (frame) a neve. Ezt a modellt hívják Többdokumentumos Interfésznek (MDI), mivel egy alkalmazás általában minden dokumentumhoz külön gyermekablakot használ. Ha a Windows Jegyzettömbjét használjuk, akkor csak egyszerre egy szövegdokumentumot tudunk megnyitni, mert ez nem MDI alkalmazás. A kedvenc szövegszerkesztőnkkel viszont több különböző dokumentumot is megnyithatunk, mindet a saját gyermekablakában, mert ez egy MDI alkalmazás. Minden egyes dokumentumra vonatkozó ablakot egy keret tartalmaz. Általánosságban elterjedt, hogy az MDI-t egy olyan eljárásnak tekintik, melynek során a felhasználó egyszerre dolgozhat több dokumentumon és fájlon, bár nem mindig ez a helyzet. A Windows 3.1-ben például a Programkezelő és Fájlkezelő arra használták az MDI-t, hogy rendszerinformációkat – nem pedig dokumentumokat – jelenítsenek meg, más alkalmazások pedig arra használják, hogy ugyanazt az adatot más-más szempontból mutassák meg különböző ablakokban.

Ez a rész rövid áttekintést ad az MDI-ről, a Windows szempontjából. Felejtsük el egy pillanatra a Delphit, és megpróbálom megmagyarázni, mi is az az MDI valójában (nem pedig azt, hogy hogyan is néz ki egy MDI applikáció). Ha még soha nem készítettünk MDI alkalmazást, és szeretnénk gyorsan belevágni, akkor érdemesebb átugornunk ezt a részt.

Megjegyzés

A Windows 3.0 megjelenésekor a Microsoft az MDI alkalmazások használatát favorizálta, de mire a Windows 3.1 megjelent, kiderült, hogy az ilyen típusú interfész sok felhasználó számára kényelmetlen. A Windows 95 megjelenésekor már érezhető volt egy újfajta hozzáállás a felhasználó felületek kialakításánál: többek között a Windows Intézője is egy SDI (Single Document Interface) alkalmazás. Mindezek ellenére nem kell még búcsút mondanunk az MDI-nek, hiszen rengeteg alkalmazás (köztük sok irodai programcsomag) ezt használja, és sok felhasználó már megbarátkozott az ilyen típusú felületekkel.

Azt tudjuk, hogy MDI alapja az, hogy van egy a fő ablakhoz hasonló gyermekablakunk, mely egy másik ablakon belül helyezkedik el. Amit még tudnunk kell az az, hogy ezen munka elvégzéséhez a Windowsnak szüksége van egy komplex szerkezetre. Ha úgy akarjuk létrehozni a gyermekablakot, hogy egyszerűen behelyezünk egy ablakot egy másikba, akkor igen furcsa dolgok történnek. Ha nem hisszük, próbáljuk meg lefuttatni a Child példát (a 15.1. ábra ennek az alkalmazásnak a kimenetét mutatja). Mikor ezt a programot futtatjuk, az ablakokkal való munka közben érdekes tapasztalatra tehetünk szert. Figyeljük meg, hogy az aktív gyermekablaknak nincsen aktív fejléce. Nem aktiválódik akkor sem, ha rákattintunk a gyermekablak kliens területére és teljes méretre hozásakor sem úgy viselkedik, ahogy azt várnánk.



15.1. ábra *A Child alkalmazás futás közben. Az ablak majdnem úgy néz ki, mint egy MDI alkalmazás, de egyáltalán nem úgy működik*

A következő példa két formot definiál: a főformot és a gyermekformot. Amikor a főformra kattintunk, egy új gyermekform jön létre a következő kóddal:

```
procedure TMainForm.FormClick(Sender: TObject);
var
  NewForm: TChildForm;
begin
  {increase the child window counter}
  Inc (Counter);

  {create a new form and define it
  as child of the current form}
  NewForm:= TChildForm.Create (self);
  NewForm.Parent := self;

  {add the number to the caption, and move it slightly}
  NewForm.Caption := NewForm.Caption + ' ' +
    IntToStr (Counter);
  NewForm.Left := (Counter * 20) mod 500;
  NewForm.Top := (Counter * 20) mod 450;

  {show the form}
  NewForm.Show;
end;
```

Ebben az eljárásban a Counter egy egész típusú private mezője a formnak, a TChildForm pedig a form adattípusa, melyet egy második egységben, a CHILDF.PAS-ban határozunk meg. Ez a második egy sima, alapértelmezésű form, különleges tulajdonságok nélkül, akárcsak a főform. A Counter értékét arra használjuk, hogy minden formnak külön nevet adjunk és hogy megakadályozzuk, hogy megjelenítéskor ugyanabba a pozícióba kerüljenek. A program kulcseleme a következő sor: NewForm.Parent := self, amely az új form Parent (Szülő) tulajdonságának a self értéket adja. Ez teszi az új formot a főablak gyermekévé.

Amint láttuk, ez a program egy MDI alkalmazásnak tűnik, de mégsem az. Az ebből adódó problémák többek között azok, hogy az aktív gyermeknek nincsen aktív fejléce, a gyermek ablakok csak úgy kerülhetnek előtérbe, ha a keretüket választjuk ki, és a szokásos módon nem hozhatók teljes méretre. Ez persze nem a Delphi hibájából adódik, hanem a Windows viselkedéséből. Ezt a helytelenül működő alkalmazást a kívánt viselkedéshez szükséges Windows üzenetekkel korrigálhatjuk, de van egy egyszerűbb megoldás is: az MDI használata által. Az MDI-t épp az ilyen problémák megoldására fejlesztették ki.

Habár az MDI szerkezete nem egyszerű, azért a programozóknak számos előnyt nyújt. Például a Windows a gyermekablakok listáját az alkalmazás egyik legördülő menüjében kezeli, továbbá vannak olyan speciális Delphi eljárások,

melyek aktiválják a megfelelő MDI funkciót, hogy elrendezze a gyermekablakokat. Az MDI alkalmazás technikai felépítése a Windowsban a következő:

- Az alkalmazás fő ablaka keretként vagy konténerként viselkedik. Ehhez az ablakhoz saját menüszerkezetre és néhány specifikus kódra van szükség (főleg az API-val történő programozás során).
- Az MDI kliensként ismert speciális ablak, a keretablak, amely az egész kliensterületét lefedi, különleges képességekkel rendelkezik, például az MDI kliens kezeli a gyermekablakok listáját. Igaz ez elsőre furcsán hangzik, de az MDI kliens a Windows egyik előre definiált kontrollja, akárcsak egy Edit Box vagy egy List Box. Az MDI kliensablak nem rendelkezik az ablakok olyan tipikus tulajdonságaival, mint a címke vagy a keret, de azért látható, sőt az MDI munkamezőjének (melynek neve: Application Background – Alkalmazás Háttér) állandó színét is meg lehet változtatni a Windows 95 Display Properties (Képernyő tulajdonságai) párbeszédablakában lévő **Appearance** (Megjelenés) oldalon.
- Az MDI alkalmazásoknak több gyermekablaka van, melyek között vannak azonosak és különbözőek. Ezek a gyermekablakok nincsenek közvetlenül a keretablakba helyezve, de mind az MDI kliensablakának gyermekeként vannak definiálva, amely pedig a keretablak gyermeke (így tulajdonképpen a gyermekablakokat a keret „unokáinak" tekinthetjük).

Amikor a Windows API segítségével programozunk, külön munkát jelent a szerkezet felépítése és karbantartása, valamint különböző kódokra van szükség magának a menünek a kezeléséhez. Amint azt a következő fejezetben látni fogjuk, ezek a feladatok könnyebbé válnak a Delphi segítségével.

Keret és gyermekablakok a Delphiben

A Delphi megkönnyíti az MDI alkalmazások kidolgozását, még az MDI alkalmazássablon használata nélkül is. Mindössze két formot kell létrehoznunk, az egyiket a FormStyle tulajdonság segítségével az fsMDIForm-hoz, a másikat ugyanezzel a tulajdonsággal az fsMDIChild-hoz beállítva. Nagyjából ez minden tennivalónk.

Az első MDI demo Delphiben

Miután a két formnak megvannak a FormStyle tulajdonsághoz tartozó értékei, el kell látnunk egy olyan lehetőséggel, mellyel létre lehet hozni egy vagy több gyermekablakot. Ez úgy is megoldható, hogy hozzáadunk egy menühöz egy New menüpontot, és beírjuk a következő kódot:

```
procedure TMainForm.New1Click(Sender: TObject);
var
  ChildForm: TChildForm;
begin
  ChildForm := TChildForm.Create (Application);
  ChildForm.Show;
end;
```

Megjegyzés

Egy új form készítésekor a Create konstruktornak átadhatjuk akár az Application értéket, akár a szülő formot (self). Az esetek döntő többségében mind a két megoldás elfogadható, de van azért néhány eltérés a két módszer között. Az átadott paraméter a form tulajdonosát fogja megadni, azt az elemet, amelyik majd megsemmisíti a formot, mikor bezárjuk. Néhány esetben a fenti kapcsolat az ablakok viselkedésére is hatással lehet (például mikor kis méretre hozunk egyet), különösen sok formmal rendelkező alkalmazásoknál.

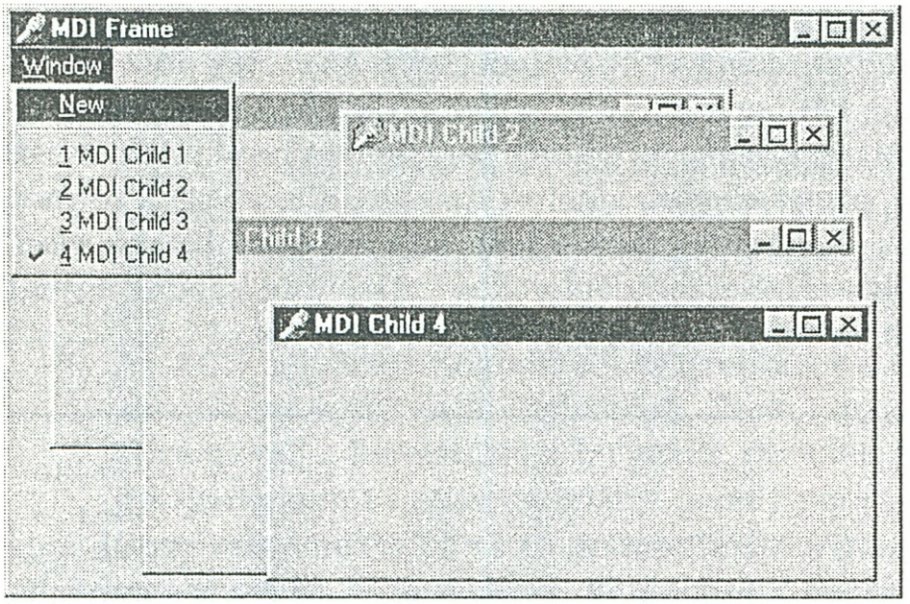
A fenti kódban két formot, a MainForm-ot, és a ChildForm-ot neveztük meg. Mivel a TChildForm osztályra hivatkozunk, az egységnek, amelyik ezt meghatározza, ebbe a forráskódba kell tartoznia. Az MDIDemo1 példában ez a második fájl a CHILD.PAS.

Hogy egy jobb programot hozzunk létre, elnevezhetjük a legördülő menüt úgy, hogy benne legyen a Window szócska, majd ezt a form WindowMenu tulajdonságának értékeként használhatjuk. Természetesen másik nevet is lehet a menütételhez választani, de a „Window" a megszokott. Ezekkel az egyszerű műveletekkel, melyekhez kevesebb mint egy perc kell, felépítettük az első MDI példaprogramot. Ahhoz, hogy valóban működjön ez a program, még néhány lépést meg kell tennünk. Először is ügyeljünk rá, hogy kezdetkor csak az első form (a főform) jelenjen meg automatikusan. Ezt beállíthatjuk a Project Options párbeszédablak **Forms** oldalán. Majd a létrehozott gyermekablakok címéhez adhatunk egy számot:

```
procedure TMainForm.New1Click(Sender: TObject);
var
  ChildForm: TChildForm;
begin
  WindowMenu := Window1;
  Inc (Counter);
  ChildForm := TChildForm.Create (self);
```

```
    ChildForm.Caption := ChildForm.Caption + ' ' +
      IntToStr (Counter);
    ChildForm.Show;
  end;
```

Az MDI példaprogram első verziója végrehajtja az MDI alkalmazás néhány általános feladatát. A 15.2. ábra példát mutat ennek a programnak a kimenetére. Számos gyermekablakot lehet nyitni, kisméretűre, illetve teljes méretűre lehet őket hozni, be lehet őket zárni, illetve a Window legördülő menü használatával válogathatunk közöttük.



15.2. ábra *Példa az MdiDemo1 program futására*

Amint a 15.2. ábrán láthattuk, a Delphi már hozzáadta a gyermekablakokat a Window menühöz, használva azok nevét. Ha több mint kilenc gyermekablakot hozunk létre, akkor egy úgynevezett More Windows menüpont adódik a legördülő menühöz; amikor ezt a menüpontot választjuk, megjelenik egy párbeszédablak (ez a Windows része, nem pedig a mi programunké) a gyermekablakok teljes listájával.

Tegyük fel, hogy be akarunk csukni néhány gyermekablakot, hogy helyre tegyük programunk kliens területét. Ha azonban a gyermekablakok Bezárás gombjára kattintunk, az ablakok kis méretűek lesznek! De hát mi történik itt? Tudjuk, hogy mikor bezárunk egy ablakot, tulajdonképpen eltüntetjük a képernyőről. A Delphiben a bezárt form, annak ellenére, hogy nem látható, továbbra is létezik! A gyermekablakok esetében az egyszerű eltüntetés nem működik, mivel az MDI Window menü és az ablakok listája továbbra is kiírja a meglévő gyermekablakokat, akkor is, ha épp el vannak rejtve, ezért a Delphi csupán kis méretűre hozza az MDI gyermekablakot, mikor azt bezárjuk. A

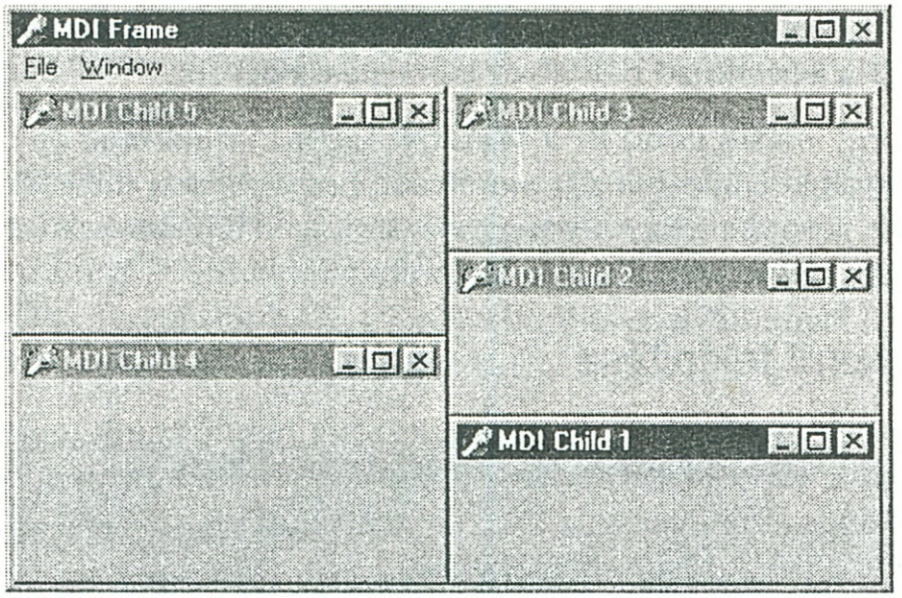
probléma megoldásaként ki kell törölnünk a gyermekablakokat, ha be akarjuk őket zárni, mint ahogy azt a következő példában is megtesszük.

Az MDIDemo1-gyel van egy másik gond is: a Window legördülő menüje egy kissé hiányos. Hozzá adnunk kell adnunk pár parancsot, hogy elrendezhessük a gyermekablakokat és az ikonokat. Ez szabványos bármelyik Windows MDI alkalmazásban, és könnyen implementálható a Delphiben.

Egy teljes Window menü felépítése

Az első feladatunk, hogy egy megfelelőbb menüszerkezetet határozzunk meg a példánkhoz. Általában a Window legördülő menünek legalább három mezője van: a Cascade (lépcsőzetes elrendezés), a Tile (mozaikszerű elrendezés) és az Arrange Icons (ikonok elrendezése). A menüparancsok kezeléséhez néhány előre definiált metódust használhatunk, melyek azon formokban érhetők el, amelyeknek a FormStyle tulajdonságban tartozó fsMDIForm érték van:

- A Cascade módszer a nyitott MDI gyermekablakokat rendezi el lépcsőzetesen. A gyermekablakok a keretablak kliens területének bal felső sarkától lefelé a bal alsó sarokig vannak elhelyezve. Az ablakok takarják, fedik egymást. Az ikon állapotban lévő gyermekablakok is el vannak rendezve (lásd lentebb az Arrange Icons címszó alatt).
- A Tile módszer az MDI gyermekablakokat mozaikszerűen rendezi el. A gyermekablakok úgy vannak elhelyezve, hogy nem fedik egymást. A keretablak kliens területe egyenlő részekre van osztva a különböző ablakok számára, így mindegyik látható a képernyőn függetlenül attól, hogy hány van belőlük. A 15.3. ábra öt gyermekablak egymás mellé helyezését mutatja. Ez a módszer az ikon állapotban lévő gyermekablakokat is elrendezi. Az alapállapot a vízszintes egymás mellé helyezés, több gyermekablak esetén azok megfelelő számú oszlopban fognak megjelenni. Ezt az alaphelyzetet a TileMode tulajdonság alkalmazásával megváltoztathatjuk.
- A TileMode tulajdonság meghatározza, hogy hogyan kell a Tile eljárásnak működnie. A két választási lehetőség közül a TBHorizontal a vízszintes, a TBVertical a függőleges elrendezés esetére használható. Egyes alkalmazások két különböző menüparancsot használnak a két egymás mellé helyezési módhoz, más alkalmazásoknál csak egy Tile parancs van, de ennek kiadásakor ügyelni kell rá, hogy a SHIFT billentyű le legyen nyomva. Ez zavarni szokta a felhasználót, ezért tanácsosabb az egyszerűbb, egy opciós alkalmazást használni.
- Az Arrange Icons eljárás az ikonállapotban lévő gyermekablakokat rendezi el, a keretablak kliens területének bal alsó sarkától kezdődően a jobb felső sarok felé haladva. A nyitott formok nem mozdulnak el.



15.3. ábra *Példa a gyermekablakok mozaikszerű elrendezésére az MdiDemo2 programban*

Ezek az eljárások és tulajdonságok egy MDI alkalmazás Window menüjének kezelésére alkalmasak. Például beírhatjuk a következő sorokat:

```
procedure TMainForm.Cascade1Click(Sender: TObject);
begin
  Cascade;
end;
```

A program többi menüparancsának kezelése is egyszerű. Vannak további hasznos módszerek és tulajdonságok, melyek szorosan kapcsolódnak a Delphiben lévő MDI-hez:

- Az ActiveMDIChild az MDI keretform futásidejű, csak olvasható tulajdonsága, mely az aktív gyermekablakot adja vissza. A felhasználó ezt az értéket megváltoztathatja úgy, hogy egy új gyermekablakot választ, vagy a programon keresztül a Next vagy a Previous utasítások alkalmazásával.
- A Next utasítás a belső sorrenden belül az éppen aktív gyermekablak rákövetkezőjét aktivizálja.
- A Previous ugyanezt teszi, csak az előző gyermekablakkal.
- A ClientHandle tulajdonság az MDI kliensablak (mely a főform kliens területét fedi) Windows leíróját tartalmazza.
- Az MDIChildCount tulajdonság a gyermekablakok aktuális számát tárolja. Az MDIChildren tulajdonság a gyermekablakok tömbje. Ezt és az MDIChildCount tulajdonságot a gyermekablakok közötti mozgásra

használhatjuk, például egy ciklus használatával, de hasznos lehet egy meghatározott gyermekablak keresésében, illetve a velük történő munka során is.

Láthatjuk, hogy a gyermekablakok belső sorrendje a működésbe hozatal sorrendjének a fordítottja, vagyis az utoljára kiválasztott gyermekablak az aktuális ablak (ez a belső listán az első), az utolsó előtti gyermekablak a második, az első pedig az utolsó a belső sorrendben. Ez a sorrend meghatározza, hogy az ablakok hogyan helyezkednek el a képernyőn. A listán lévő első ablak az összes többi fölött van, míg az utolsó a többi alatt, talán el is tűnik. Képzeljünk el egy geometriai tengelyt (a Z tengelyt), mely a képernyőből kifelé mutat a mi irányunkba. Az aktív ablak egy magasabb Z koordinátaértékkel bír, így takarja a többi ablakot. Ezért a Windows sorrend sémája z-sorrend néven ismeretes.

A gyermekablak OnClose eseményéhez pár sornyi kódot hozzá kell írnunk, hogy azok kiírása megfelelően működjön, mint azt már korábban is említettük:

```
procedure TChildForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;
end;
```

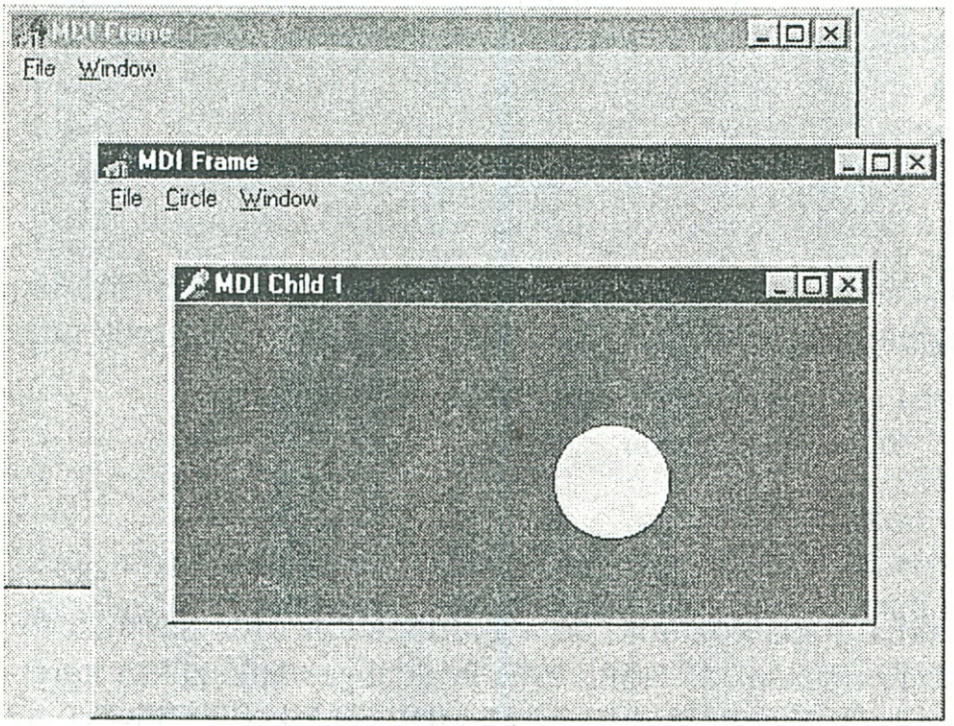
Mostanáig a keretablakkal foglalkoztunk, de a gyermekablakhoz használt főformnak nincsenek komponensei, és csak kevés kóddal rendelkezik. A keretablak általában nem sokat változik a különböző MDI példákban (kivéve az eszköztárat, a státuszsort és hasonló eszközöket). Az esetek többségében a kódok a gyemekformokba kerülnek. Most pedig itt az ideje, hogy megnézzünk egy igazi példát a gyermekablak használatára.

Egy gyermekablak létrehozása

Az utóbbi két példában megfigyeltük, hogy hogyan lehet a Delphiben egy MDI alkalmazást létrehozni úgy, hogy a keretablakra összpontosítottunk. Így egy tipikus MDI tulajdonságokkal rendelkező programot kaptunk, ami viszont nem túl hasznos. Mit kezdhetünk egy gyermekablakkal? A válasz: bármit, amit egy formmal. Számos összetevőt adhatunk hozzá, szerkesztőket hozhatunk létre, grafikai programokat tehetünk hozzá és így tovább. Bármelyik program, melyet eddig megszerkesztettünk, átalakítható MDI alkalmazássá (bár egyes esetekben ez értelmetlen lenne).

Az első példa egy egyszerű grafikai program MDI verziója (hasonló a Shape program második verziójához, melyet a 10. fejezetben készítettünk), melynek neve: MDIDemo3. Ez a program egy kört tud megjeleníteni azon a helyen, ahová az egér egyik gombjával kattintunk. A 15.4. ábra példát mutat az MDIDemo3 alkalmazására. A programnak van egy Circle nevezetű menüje,

mely lehetővé teszi, hogy megváltoztassuk a kör felszínének a színét, vagy a körvonal színét, illetve vastagságát. Az egésznek az az érdekessége, hogy a gyermekablak programozásakor nem kell törődnünk más formok, és a keretablak jelenlétével. Egyszerűen megírjuk a form kódját és kész. Egyedül a két form menüjére kell odafigyelni.



15.4. ábra *Az MdiDemo3 példa két példánya futás közben, az egyikben egy kört tartalmazó gyermekablakkal és a dinamikus menüvel. Figyeljük meg, hogy a két példány menüsora között különbség van!*

Mikor főmenüt készítünk a gyermekform számára, annak, ha megjelenik a képernyőn, onnan kezdve a keretablak fő menüjét kell helyettesítenie. Egy MDI gyermekablaknak tulajdonképpen nem lehet saját menüje, de ez ne zavarjon bennünket, mivel ez az MDI alkalmazás általános jellemzője. Használhatjuk a keretablak menüsorát a gyermekablak megjelenítéséhez. Sőt, sokkal jobb, ha összevonjuk a két menüsort. Például ebben a programban a gyermekform menüje elhelyezhető a keretablak File és Window legördülő menüi között. Ezt a következő GroupIndex értékekkel lehet végrehajtani:

- File legördülő menü, főform: 1
- Window legördülő menü, főform: 3
- Circle legördülő menü, gyermekform: 2

Amikor ezeket a beállításokat használjuk a menücsoport indexekhez, a keretablak menüsorában két vagy három legördülő menü lesz. Kezdetben a menüsornak két menüje van. Amint létrehozunk egy gyermekablakot, három menünk lesz és mikor az utolsó gyermekablakot bezárjuk (kitöröljük), a Circle legördülő menü eltűnik. Ezt a 15.4. ábrán láthatjuk, de érdemes megvizsgálni úgy is, hogy lefuttatjuk a programot.

A főform forráskódja ugyanaz, mint a program előző verziójáé. A gyermekablak kódja csak kis mértékben változott a 9. fejezethez képest. Ha megnézzük a forráskódot (a mellékelt lemezen), megfigyelhetjük, hogy a futó program menüparancsai hogyan kapcsolódnak a két formhoz, és hogy a forráskódban minden form a saját parancsait kezeli függetlenül más tényezők jelenlététől.

A gyermekform adatait, kiváltképp a kör középpontjának koordinátáit a form mezőinek alkalmazásával kell megadni, nem pedig egyéb, a uniton belüli változókkal. Sőt, szükség lesz egy külön memóriahelyre, ahol a gyermekablakokhoz tartozó körközéppontokat tároljuk.

Megjegyzés

A gyermek formok adatainak tárolása egyszerű Delphiben, de már jóval bonyolultabb Windows API-t használva. A hagyományos Windows programok komplex struktúrákban tárolják az MDI alkalmazások ablakainak adatait. Nem szeretnék részletekbe bocsátkozni, de nem árt tudnunk, milyen nagy előnyökkel jár az objektum-orientált eszközök használata egy olyan operációs rendszernél, amely valójában nem objektum-orientált.

MDI alkalmazások különböző gyermekablakokkal

A komplex MDI alkalmazások készítésének általános megközelítési módja az, hogy különböző fajtájú (vagyis eltérő gyermekformokon alapuló) gyermekablakok szerepelnek benne. Felhasználhatjuk az előző példát, hogy rávilágítsunk néhány problémára, melyet ez a megközelítés okozhat. Ehhez a példához egy új gyermekformot kell készítenünk. Bármilyen form megfelelne, de ezzel a példával valami újat szeretnék mutatni: a korlátozott párhuzamos folyamatokat. Ezért elhatároztam, hogy olyan formot használok, melyben egy pattogó négyzet van.

Egy pattogó alak hozzáadása

A négyzet, egy Shape komponens, a form kliens területén halad körbe meghatározott időközönként, egy Timer komponenst használva, és irányát váltogatva a form egyik szélétől a másikig pattog. Ezt a mozgatási folyamatot egy meglehetősen összetett algoritmus határozza meg (legalábbis a könyv többi példájához képest). Az elgondolás az, hogy a négyzetnek megvan a maga pozíciója, valamint egy Dir (Direction, azaz Irány) értékkel – ami a form osztály egy másik tagja – van összefüggésben, amely a következő értékek egyikét veheti fel:

```
type
  Directions = (up_right, down_right,
    down_left, up_left);
```

Mikor egy bizonyos idő eltelik, a négyzet a megfelelő irányba halad:

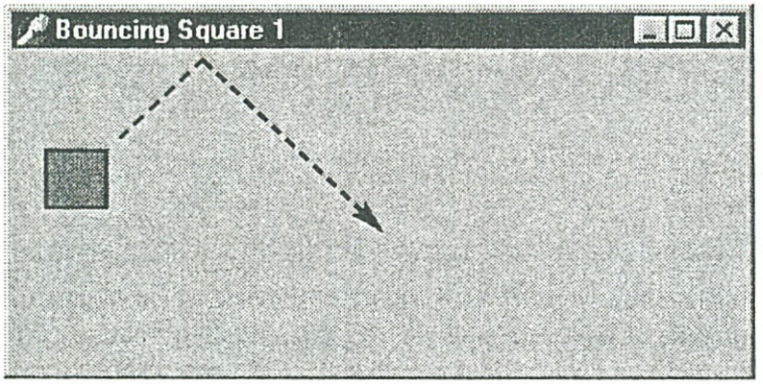
```
procedure TBounceChildForm.Timer1Timer(Sender: TObject);
begin
  case Dir of
    up_right: begin
      Shape1.Left := Shape1.Left + 3;
      Shape1.Top := Shape1.Top - 3;
    end;
    ...
```

Ez a rész felelős a mozgásért. A feladat az, hogy a négyzetet a form szélein pattogtassuk, vagyis: minden esetben, mikor a négyzet elér az egyik szélre, meg kell változtatnunk mozgásának irányát. Ahhoz, hogy meghatározzuk, hogy mikor ér az alakzat a szélre, leellenőrizhetjük, hogy a felső, és bal oldali értékei nullát, az alsó és jobb oldaliak pedig az ablak szélének koordinátáit veszik-e fel. Az alsó és a jobb oldali értékek nem érhetők el közvetlenül, de kiszámíthatók ha a felső értékhez hozzáadjuk az alakzat magasságát (vagy a szélességet a baloldali értékhez).

```
if Shape1.Top <= 0 then ...
if Shape1.Top + Shape1.Height >= ClientHeight then ...
if Shape1.Left <= 0 then ...
if Shape1.Left + Shape1.Width >= ClientWidth then ...
```

Megpróbálhatjuk elvégezni ezeket az ellenőrzéseket a kód végén azért, hogy növeljük az aktuális értékeket, de nem fog sikerülni. Sőt, ahhoz, hogy a négyzet pontosan mozogjon, egy megfelelő (jóval összetettebb) modellt kell választani. Például azt is szeretnénk, hogy minden fordulat 90 fokos legyen, hogy miután a négyzet eléri a keretet, ne forduljon vissza ugyanabba az irányba, amerről érkezett (15.5. ábra).

Ha a form maga egy négyzet, akkor az alakzat mindig az óra járásával megegyező vagy azzal ellentétes irányba fog haladni, a kezdeti utasítástól függően. Ha a form téglalap alakú, a kétféle mozgás váltakozik (15.5. ábra). Hogy



15.5. ábra *A Shape objektum pontos útvonala, amelynek 90 fokkal kell irányt váltania, valahányszor a keretnek ütközik*

elérjük ezt a viselkedést, be kell írnunk néhány programsort, hogy megváltoztassuk a meglévő irányokat a már említett eset mindegyikében. Sőt, a négyzet következő iránya (miután elérte a form egyik szélét) midig az éppen aktuális iránytól függ. Az ennél a formnál kulcseljárásnak számító, meglehetősen hosszú Timer1Timer módszer első része a következő:

```
procedure TBounceChildForm.Timer1Timer(Sender: TObject);
begin
  case Dir of
    up_right:
    begin
      Shape1.Left := Shape1.Left + 3;
      Shape1.Top := Shape1.Top - 3;
      if Shape1.Top <= 0 then
        Dir := down_right;
      if Shape1.Left + Shape1.Width >= ClientWidth then
        Dir := up_left;
    end;
    ...
```

Az új gyermekform menüje

Mint bármelyik gyermekformnak, a mi példánkban szereplőnek is saját menüsora van, amely össze van kapcsolva a keretablak menüsorával. A menü szerkezete meglehetősen egyszerű, habár két különböző legördülő menüje is van. Az első legördülő menüben (Square) két parancs van a négyzet színének megváltoztatására, illetve pozíciójának megállapítására. Az első módszer kódja egyértelmű, a második pedig a következő:

```
procedure TBounceChildForm.GetPosition1Click(
  Sender: TObject);
begin
  MessageDlg (
```

```
        'The top-left corner of the square was in the
         position (' + IntToStr (Shape1.Left) + ', ' +
         IntToStr (Shape1.Top) + ').', mtInformation,
         [mbOk], 0);
end;
```

Mivel a párbeszédablak színválasztását egyetlen színre használjuk (az egész négyzet ilyen színű) ezért ezt nem kell egy külön változóban tárolni. Persze attól még alapértékként meg kell határoznunk a kezdő utasításokkal együtt, a FormCreate eljárás során.

A második legördülő menü (Movements) két parancsa: elindítani vagy megállítani az alakzat mozgását. Ez az időmérő elindításával és kikapcsolásával oldható meg. Emellett mindkét módszer hatályon kívül helyezi a megfelelő parancsot, és aktivizálja annak ellenkezőjét, ugyanis értelmetlen egy mozgó alakzatot beindítani, vagy egy már állót leállítani. Íme az első:

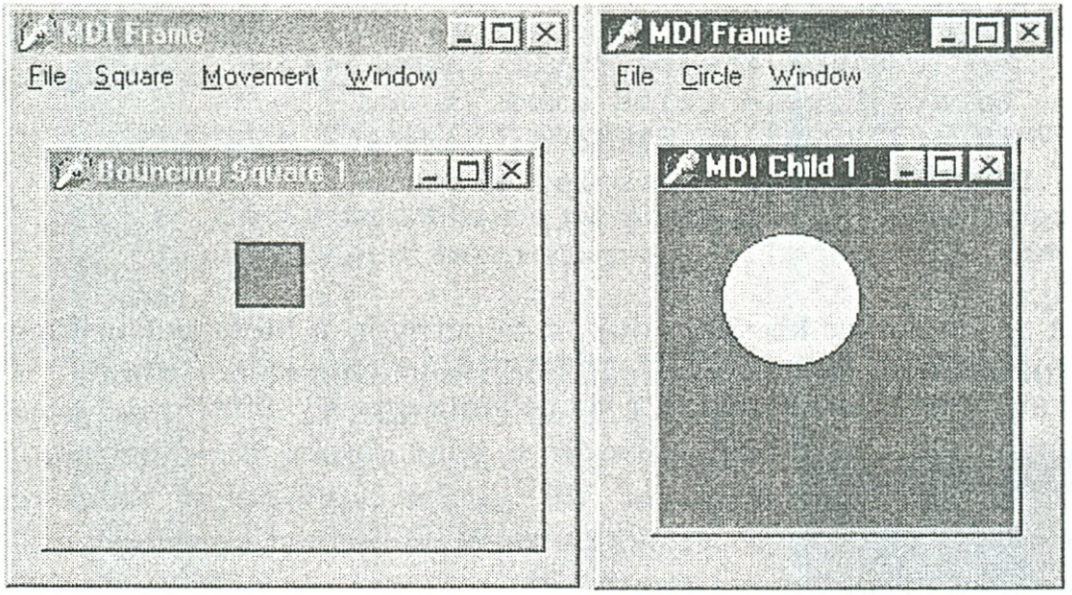
```
procedure TBounceChildForm.Start1Click(Sender: TObject);
begin
  Timer1.Enabled := True;
  Start1.Enabled := False;
  Stop1.Enabled := True;
end;
```

A főform átalakítása

Miután a pattogó négyzet formja kész, be kell illeszteni azt az MDI alkalmazásba. A főformnak olyan menüparanccsal kell rendelkeznie, mely lehetővé teszi egy ilyen gyermekform létrehozását, valamint a legördülő menük csoportindexeinek ellenőrzését. Ennek a formnak a menüszerkezetét megváltoztattam. A File legördülő menünek most van egy New menüpontja, amely az újfajta gyermekablak létrehozását végzi. A kód ugyanazt a gyermekablak számlálót használja, mint a másik form. Lehetséges a kétféle gyermekablakhoz két különböző számláló használata is.

Amint egy ilyen form megjelenik a képernyőn, a menüsora automatikusan összekapcsolódik a fő menüsorral. Mikor a kétféle gyermekform közül kiválasztunk egyet, a menüsor értelemszerűen megváltozik. Ha az összes gyermekablakot bezárjuk, a főform eredeti menüsora jelenik meg. A megfelelő menücsoport indexek használatával hagyjuk, hogy a Delphi mindent automatikusan hajtson végre, amint az a 15.6. ábrán lévő két ablakban látható.

További menüpontokkal bővítettem a fő formot. Az egyik menüválasztással bezárunk minden gyermekablakot, egy másik vízszintesen helyezi el a gyermekablakokat függőleges helyett, egy további pedig adatokat mutat róluk. Ezen menüpontokhoz kapcsolódó eljárások a form néhány külön MDI tulajdonságát használják, úgy, mint a TileMode, az MDIChildCount és az MDIChildren tömb.



15.6. ábra *Az MdiDemo4 program menüsora az éppen aktuális gyermekablaktól függően változik*

Ezek közül kettő nagyon érdekes, mivel megmutatják, hogyan lehet a gyermekablakok mindegyikén dolgozni. Például a CloseAll1Click eljárás kódja a következő:

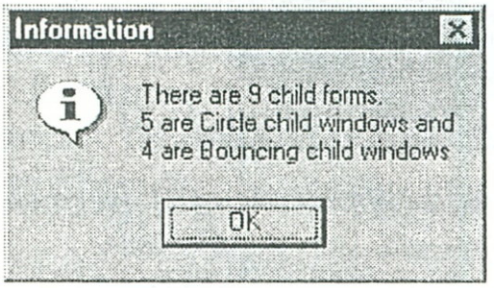
```
procedure TMainForm.CloseAll1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to MDIChildCount - 1 do
    MDIChildren [I].Close;
end;
```

Egy másik példával a Count parancshoz tartozó módszernél találkozhatunk. Ebben az esetben a tömböt végignézzük, hogy megszámoljunk minden gyermekablakot. Ezt az `is` nevű RTTI operátor hajtja végre:

```
procedure TMainForm.Count1Click(Sender: TObject);
var
  NBounce, NCircle, I: Integer;
begin
  NBounce := 0;
  NCircle := 0;
  for I := 0 to MDIChildCount - 1 do
    if MDIChildren is TBounceChildForm then
      Inc (NBounce)
    else
      Inc (NCircle);
```

```
      MessageDlg (
        Format ('There are %d child forms.'#13 +
          '%d are Circle child windows and ' +
          '%d are Bouncing child windows',
          [MDIChildCount, NCircle, NBounce]),
        mtINformation, [mbOk], 0);
    end;
```

Miután ezeket az értékeket kiszámítottuk, a képernyőn a MessageDlg függvénnyel jeleníthetők meg, amint azt a fenti kódban láthatjuk (15.7. ábra).



15.7. ábra *Az MdiDemo4 példa Count menüparancsának eredménye, amely megmutatja, hogy hány, és milyen típusú gyermekablakok tartoznak a programhoz*

Az MDI gyors elsajátítása

Ebben a fejezetben alaphelyzetből építettünk fel MDI alkalmazásokat a Delphiben. Ezt a megközelítést alkalmazva részletesen tanulmányozhattuk az MDI-t a Windowsban, valamint a Delphi közreműködését az eljárás során. Azonban az alaphelyzetből való kiindulás nem a legjobb módja egy igazi MDI alkalmazás létrehozásának. Ehhez a Delphi biztosít egy MDI Application sablont, továbbá használhatjuk valamelyik Varázslót, hogy egy MDI alkalmazás kezdőkódját gyorsan elkészítsük. Először megnézzük ezeknek a segédprogramoknak a vázát, majd azt, hogyan készítsünk alkalmazást a Delphi által nyújtott kód segítségével.

Az MDI Application sablon használatához adjuk ki a Delphi környezet **File | New** parancsát, válasszuk a **Projects** oldalt, majd az MDI Application-t. Válasszunk egy könyvtárat a projectnek, és a Delphi átmásolja ide egy egyszerű MDI alkalmazáshoz szükséges fájlokat, az eszköztárral, a státuszsorral, és a menüvel együtt (az Olvasó kényelmének érdekében összefoglaltam a jelen fejezet MdiTemp könyvtárában lévő fájlokat a felkínált futtatható fájlokkal együtt).

A sablon nyújtotta form állandó elemei a következők: a menüsor, az eszköztárként használt Panel összetevő, egy StatusBar komponens és néhány rendszerpárbeszédablak. A form forráskódjának van néhány érdekes eleme. Azt

javaslom, készítsük el ezt a kódot, és tanulmányozzuk figyelmesen. Először figyeljük meg a FormCreate metódus kódját:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnHint := ShowHint;
  Screen.OnActiveFormChange := UpdateMenuItems;
end;
```

Az első utasítás az alkalmazási szinten lévő tippeket aktivizálja, amint azt a 11. fejezetben láttuk. Ez együttműködik a ShowHint eljárással, mely a StatusBar komponensben jeleníti meg a tippeket. A FormCreate metódus második utasítása újdonságot jelent. Mi az a Screen objektum? Mi az az OnActiveFormChange? Emlékezhetünk rá, hogy korábbi fejezetekben már használtuk a Screen általános objektumot egyes programokban: ez az objektum hozzáférést biztosít a programban beállított betűtípusok listájához. A Screen objektum számos, az alkalmazás kimenetelével kapcsolatos információt kezel, beleértve a felhasználható betűtípusokat, és az alkalmazásban lévő formok listáját. A 25. fejezetben további részletekkel ismerkedünk meg a TScreen osztállyal kapcsolatban. Pillanatnyilag elég annyit tudnunk, hogy az OnActiveFormChange esemény mindig akkor következik be, mikor az alkalmazás aktív formja megváltozik.

Az OnActiveFormChange az alkalmazás aktív formjára vagy ablakára, nem pedig magára az aktív alkalmazásra vonatkozik. Vagyis ez az eset nem aktiválódik, ha az éppen futó Windows alkalmazások között változtatjuk az aktív ablakot; ez csak akkor történik meg, ha a pillanatnyi alkalmazás aktív formját változtatjuk meg. Ez tehát egy alkalmazásra, nem pedig az egész rendszerre kiterjedő esemény. Az aktív Windows alkalmazás változtatásakor történő kódok végrehajtásához használhatjuk az Application objektum OnActivate eseményét. Ezt ne keverjük össze egy form OnActivate eseményével, mikor is maga a form aktiválódik. Hasonló eredmény eléréséhez igénybe vehetjük a Screen objektum OnActivateFormChange eseményét bármelyik form OnActivate eseménye helyett.

Ahogy az Application objektum eseményeit, úgy a Screen objektumét is meg kell adni a programban. Figyeljük meg, hogy ezt a műveletet úgy érvényteleníthetjük, ha a FormDestroy eljáráson belüli eseményt NIL-re állítjuk. Egyébként az UpdateMenuItems eljárás hatására aktivizálódik vagy kikapcsolódik néhány menüpont, mikor már nincs több gyermekablak.

```
procedure TMainForm.UpdateMenuItems(Sender: TObject);
begin
  FileCloseItem.Enabled := MDIChildCount > 0;
  FileSaveItem.Enabled := MDIChildCount > 0;
  CutItem.Enabled := MDIChildCount > 0;
  CopyItem.Enabled := MDIChildCount > 0;
```

További érdekesség a Delphi kínálta forráskódban az, ahogy a fájl megnyitását végzi. Létezik egy úgynevezett CreateMDIChild módszer, mely az új gyermekformok létrehozásáért felelős, valamint azok paramétereinek felhasználásával elnevezi őket.

```
procedure TMainForm.CreateMDIChild(const Name: string);
var
  Child: TMDIChild;
begin
  { create à new MDI child window }
  Child := TMDIChild.Create(Application);
  Child.Caption := Name;
end;
```

Ezt a módszert meg lehet hívni a File menüben mind a New, mind pedig az Open parancsokhoz kapcsolódó eljárásokkal.

```
procedure TMainForm.FileOpenItemClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    CreateMDIChild(OpenDialog.FileName);
end;
```

A fájlok nem nyílnak meg automatikusan: csak a fájlnevek kerülnek használatba. Ezt úgy fejleszthetjük tovább, hogy bemásoljuk ebbe a programba a Notes példa kódját, a RichNote példát, vagy a képnézegető példáját.

Hogyan tovább?

Számos olyan dolog van, amit megvizsgálhatnánk a Windowsban az MDI-vel kapcsolatban, de úgy gondolom, ez a fejezet elég az áttekintéshez. Úgy döntöttem, nem mutatok be több MDI példát, mivel e modell jelentőssége csökken a Windows 95-ben. Ebben a fejezetben összefoglaltuk a legfontosabb tudnivalókat a multi-formos alkalmazásokról, melyet a 13. fejezetben kezdtünk, ahol másodlagos formokat (modális és nem-modális) és párbeszédablakokat vizsgáltunk. Később láthattunk notebookon alapuló alkalmazásokat, formszétválasztó technikákat és ebben a fejezetben az MDI-t. Egyes esetekben ezek a technikák hasonló eredményességgel alkalmazhatók, tehát nekünk kell eldönteni, hogy melyiket választjuk.

Most pedig áttérhetünk egy nagyon izgalmas Delphi programozói témára: az adatbázis-kezelésre. Erről fog szólni a következő két fejezet, mely megpróbálja bemutatni a Delphi adatbázis-kezelésének alapvető összefüggéseit. Erről akár egy külön könyvet is lehetne írni, így a bemutatás nem lesz teljes, de azért egy átfogó képet fogunk kapni a Delphi ezen kulcsfontosságú elemeiről.

Ezután a két adatbázisokkal foglalkozó fejezet után végre a Delphi színfalai mögé tekinthetünk, és olyan témákra összpontosíthatjuk figyelmünket, mint a Delphi összetevők felépítése, és az ActiveX kontrollok.

16. FEJEZET

Adatbázis-kezelő alkalmazások készítése

E fejezet tartalma:

- Adatok, fájlok, adatbázisok és táblák
- A Delphi adatbázis komponensei
- Adatbázis-kezelő alkalmazások készítése manuálisan
- Az adatmezők elérése
- Mezők használata a tábla változtatásához
- Egy tábla oszlopának szummázása
- Az adatbázis különböző tábláinak használata
- Tábla létrehozása
- Többrekordú rács használata
- Kapcsolttáblás form készítése varázslóval

Adatbázis-kezelő alkalmazások készítése

Az adatbázis-kezelés a Delphi környezet egyik kulcsfontosságú eleme. Sok programozó idejének nagy részét ilyen típusú programok készítésével tölti, hiszen ezek az alapjai a napjainkban igen elterjedt adatbázis-kezelő alkalmazásoknak. Ez a fejezet betekintést nyújt a Delphi adatbázisokkal kapcsolatos lehetőségeibe, megtanuljuk, hogyan készíthetünk bonyolult alkalmazásokat egy üres formból kiindulva, vagy a Delphi Adatbázis Form Varázslója segítségével.

Arról viszont nem lesz szó, hogyan kell adatbázisokat megtervezni. Feltételezem, hogy az Olvasó már tisztában van a téma elméleti alapjaival, és már tervezett, és készített adatbázisokat. Külön adatbázis-specifikus problémákról sem fog szó esni, a célom az, hogy elmagyarázzam, hogyan működik ez a fajta programozás a Delphiben.

Először is azt fogjuk megnézni, hogyan működik a Delphiben az adatelérés, majd végignézzük a környezet adatbázisokkal kapcsolatos komponenseit, néhány egyszerű példán keresztül mutatva be azok működését. Ezután bonyolultabb műveletek következnek majd, megtanuljuk, hogyan kérhetünk információkat a táblákról futási időben, hogyan hozhatunk létre Delphi programban új táblákat, hogyan használhatunk grafikus mezőket, és hogyan építhetünk több táblával rendelkező formokat.

Megjegyzés

A Delphi Client/Server Suite kiadása a helyi adatbázisok mellett különböző SQL kiszolgálókhoz is képes csatlakozni. Erről, és más, összetettebb problémákról a következő fejezetben esik majd szó.

Adatok, fájlok, adatbázisok, és táblák

A számítógép az adatokat – beleértve az adatbázisokat is – mindig fájlokban tárolja. Ennek a tárolásnak a megvalósítására persze több mód is létezik: lehet egy egész adatbázist egyetlen állományban tartani, de felbonthatjuk azt táblákra, indexekre, vagy bármi más összetevőre, és tárolhatjuk külön fájlokban (de azért általában egy könyvtárban) a részeket.

A Delphiben mindkét megközelítést használhatjuk, pontosabban magunk alakíthatjuk ki ezt a megközelítést, amely aztán összhangban fog működni az általa elfedett struktúrákkal. Mindig hivatkozhatunk az adatbázisra a nevével, vagy egy úgynevezett álnévvel (angolul alias-szal), ami egyfajta beceneve az adatbázisnak, de ez a hivatkozás nem csak egy fájlt, hanem akár egy teljes

könyvtár tartalmát jelentheti. Mindez csupán attól függ, milyen adatformátumot használunk. A Delphi előre nem meghatározott adatbázis-szerkezettel dolgozik: használhatunk dBASE vagy Paradox táblákat, ismeri az SQL-t (Structured Query Language, vagyis Strukturált Lekérdezőnyelv), és a Microsoft ODBC (Open Database Connectivity) segítségével rengeteg típusú adatbázishoz hozzá tud férni.

Tipp

A Database Explorerrel vagy a Database Engine Configuration segédprogrammal létrehozhatunk új álneveket (alias) is. Ezt a műveletet programból is végrehajthatjuk, a BDE `DbiAddAlias` eljárásával.

A Delphi adatbázis-alkalmazások nem közvetlenül kommunikálnak az adatforrásokkal, hanem a Borland Database Engine (BDE) segítségével, amely a megfelelő meghajtóprogram használatával közvetlen kapcsolatot tud kiépíteni dBASE, Paradox, ASCII, FoxPro, Access, és más táblákkal.

Új!

A Delphi 3-ban a Microsoft Visual FoxPro és a Microsoft Access táblák támogatása újdonságnak számít, erre a korábbi verziók nem voltak képesek.

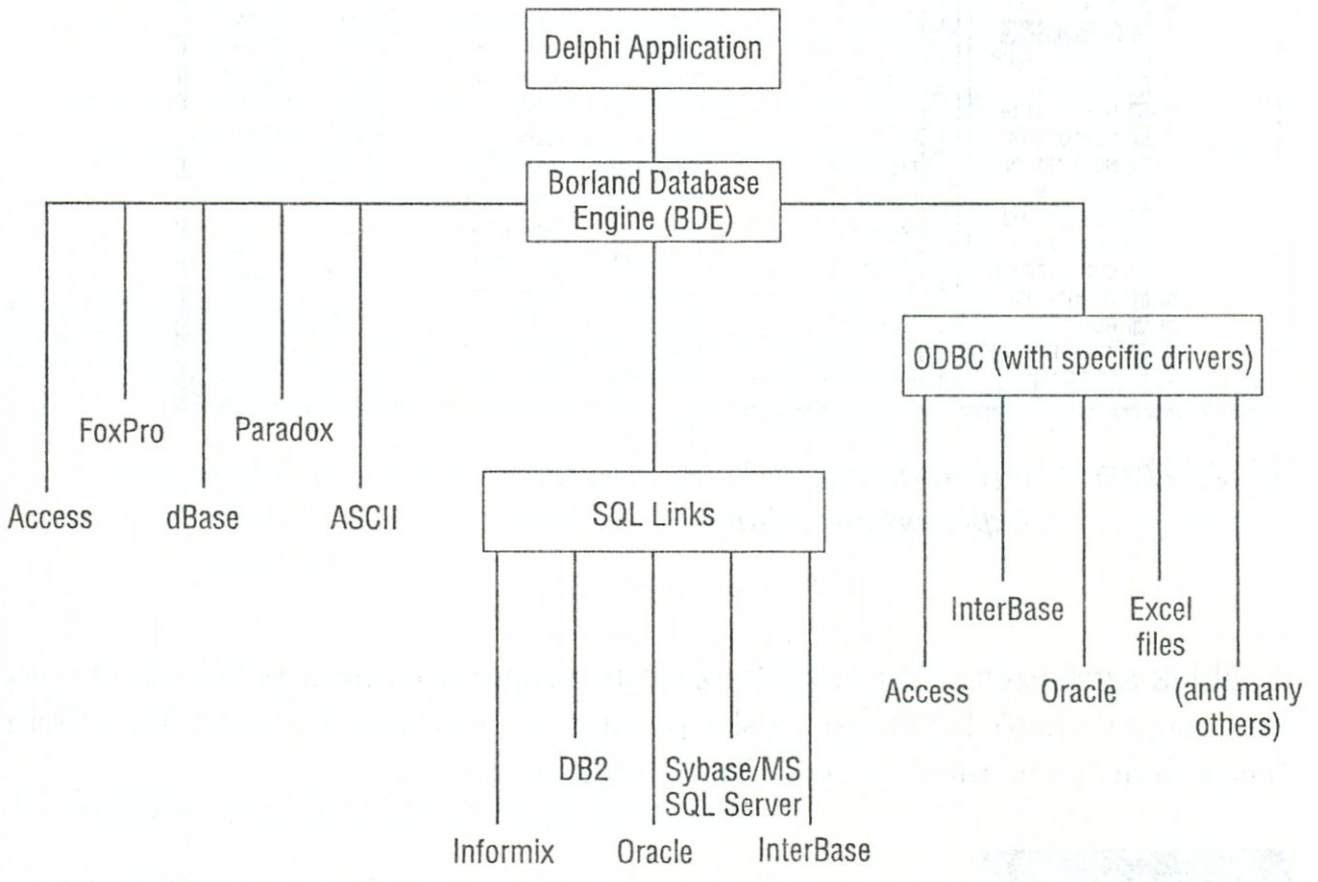
A BDE a Borland SQL Links programjával is kapcsolatban van, ennek segítségével érhet el néhány helyi és távoli SQL kiszolgálót (itt jegyezném meg, hogy a Borland cég a Local InterBase for Windows rendszerével együtt szállítja a Delphit). A távoli kiszolgálók között olyanokat találunk, mint az Oracle, a Sybase, az Informix, az InterBase, és a DB2. Ha más típusú adatbázishoz szeretnénk kapcsolódni, akkor a BDE ODBC kapcsolatát kell használnunk. Ez ugyan általában egyszerű megoldáshoz vezet, mégis óvatosan bánjunk vele, mert nem túl hatékony módszer. Csak akkor folyamodjunk az ODBC meghajtókhoz, ha nincs más választásunk! A 16.1. ábrán láthatjuk, hogyan is működik az adatok elérése a Delphiben.

Az a tény, hogy a Delphi alkalmazások a BDE-n keresztül kommunikálnak az adatbázisokkal, azt a következményt vonja maga után, hogy a programot –

alkalmazásunkkal együtt – telepíteni kell arra a számítógépre is, ahol rendszerünk futni fog. Szerencsére ez nem nehéz dolog, hiszen a Delphiben megtalálható egy telepítés-előkészítő program (InstallShield), amely segítségével telepítő lemezeket készíthetünk, amelyek a BDE-t, és programunkat is tartalmazni fogják. A BDE fájlok tehát mindenképpen szükségesek programunk futásához, de szabadon terjeszthetjük őket.

Új!

A Delphi 3-ban több olyan adatbázis-kezeléshez használt eszközt is találhatunk, amelyek nem használják a BDE-t. A rendszer tartalmaz egy ClientDataSet nevű komponenst, amely különböző gépeken futó OLE kiszolgálókhoz tud kapcsolódni. Várható az is, hogy a piacon megjelenik majd több olyan egyszerűbb meghajtóprogram, amely a BDE-t helyettesítik majd bizonyos adatbázis típusoknál. Korábban is léteztek már ilyen eszközök a Delphiben, de sokkal kevésbé voltak a VCL-be integrálva.

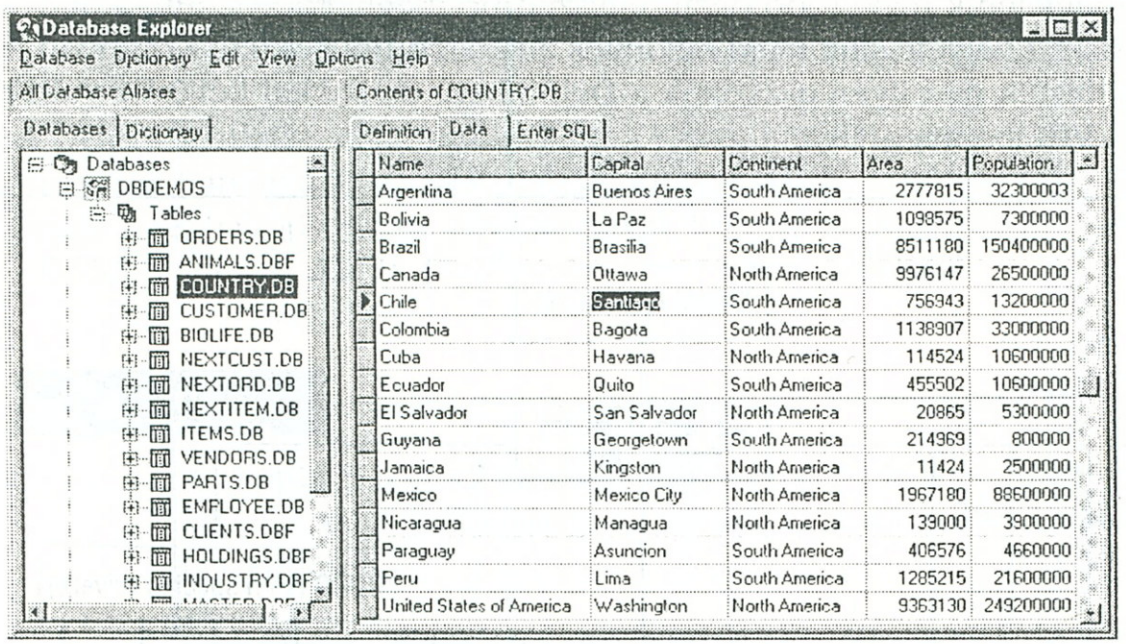


16.1. ábra *Így néz ki az adatelérés Delphiben*

Mi is egy tábla?

Általában, ha adatbázisról beszélünk, akkor táblák gyűjteményére gondolunk. De mi is egy tábla pontosan? Legtöbbünk valószínűleg tudja a választ, de az adatbázis-kezelésben járatlanok kedvéért gyorsan összefoglalom az ide vonatkozó alapvető tudnivalókat.

Az adatbázis egy táblája valami olyasmi, mint egy rekordokból álló állomány a Pascalban. Egy táblának vannak rekordjai (vagy más néven sorai), és oszlopai, pontosan annyi, ahány mezője van a rekordnak. A 16.2. ábra egy tábla felépítését mutatja, a megjelölt kulcsösszetevőkkel. Figyeljük meg a tábla használatánál felmerülő aktuális rekord (ez az a rekord, amiben a felhasználó éppen dolgozik), és aktuális mező (az aktuális rekord éppen használt mezője) fogalmakat!



16.2. ábra *Egy adatbázis táblájának sémája, ahogy az a Database Explorerben látható*

A táblák szerkezetét jól mutatják az olyan programok, mint például a Delphiben megtalálható Database Desktop, vagy a Database Explorer. Ez utóbbit használtam én is, mikor a 16.2. ábrát készítettem.

Megjegyzés

A 16.2. ábrán látható tábla a COUNTRY.DB nevet viseli, és a DBDEMOS álnév alatt található a Delphiben. A példáim nagy része ezeket a Delphivel szállított táblákat fogja használni, de megmutatom majd azt is, hogyan kell új adatbázisokat létrehozni.

A tábla első sora mindig a különböző mezőket mutatja, olyanokat, mint a példában a Name (név), Capital (főváros), Continent (kontinens), és így tovább. Azt is megfigyelhetjük a képen, hogy hogyan van ábrázolva grafikusan az éppen használt sor és oszlop.

Műveletek végzése az adatbázison

Ha megvan az adatbázisunk, akkor elkezdhetünk műveleteket végezni adatainkkal: szerkeszthetjük a meglévő értékeket, új rekordokat szúrhatunk be, vagy meglévőket törölhetünk a táblákból. Műveleteket végezhetünk olyan ablakokon is, amelyek adataink egy részét tartalmazzák, majd visszamásolhatjuk a megváltozott elemeket az adatbázisba. Erre az utolsó két lépésre több példát is láthatunk majd a későbbiek során.

Az adatbázis valódi adatai és a felhasználó által látott értékek közti szinkronizáció nem egyszerű feladat, különösen többfelhasználós környezetben, ahol egyszerre többen írhatják/olvashatják az adatokat. Az ütközések elkerülése érdekében az adatbázisoknak különböző zárolási mechanizmusai vannak, amelyek meggátolják azt, hogy egyszerre két felhasználó akarja megváltoztatni ugyanazt az adatot. Szerencsére Delphi programozóként ritkán kell ezzel a gonddal foglalkoznunk, hiszen a BDE, amelyen keresztül a műveleteket végezzük, elrejti az ilyen mélységű részleteket a számunkra.

A Delphi tehát egy egységes adatbázis-kezelő felületet biztosít számunkra, de nem szabad elfelejtenünk, hogy nem minden adatbázis rendelkezik ugyanazokkal a tulajdonságokkal.

Megjegyzés

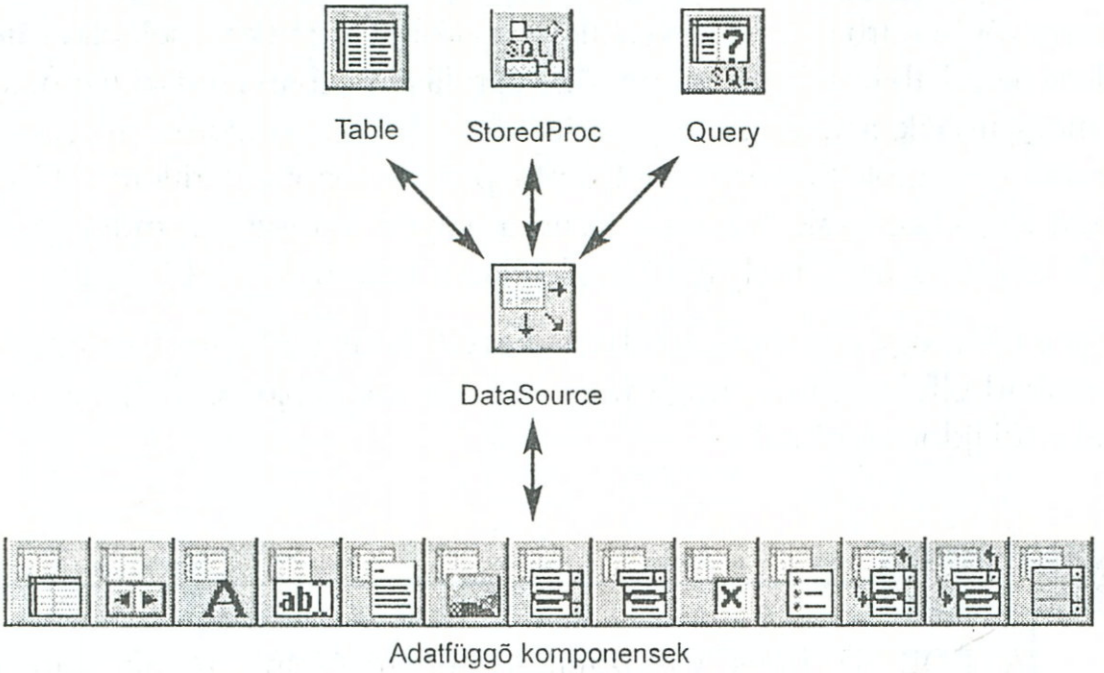
A BDE 32 bites változatában lehetőségünk nyílik tranzakciók (adatbázis-műveletek csoportosítása), és azok visszavonásának végzésére. Az SQL kiszolgálók ezen tulajdonsága most már helyi adatbázisokra is használható.

A Delphi adatbázis komponensei

A Delphi több adatbázisokhoz kapcsolódó komponenssel is rendelkezik. A komponenspaletta Data Access (adatelérés) oldalán találhatjuk azokat az összetevőket, amelyek segítségével használhatjuk az adatbázisokat. Legtöbbjük nem látható komponens, hiszen adatkapcsolatokat, táblákat, lekérdezéseket, vagy hasonló elemeket tartalmaznak. Szerencsére a Delphi számos olyan

eszközt is tartalmaz, amelyekkel láthatóvá, és ez által közvetlenül szerkeszthetővé tehetjük adatbázisainkat. Ezeket az elemeket a Data Controls oldal biztosítja számunkra. Az itt található elemeket *adatfüggő* vezérlőknek nevezzük.

A Delphiben egy adatbázis eléréséhez mindenekelőtt szükségünk van egy adatforrásra, amelyet a DataSource (adatforrás) komponenssel adhatunk meg. Ez az elem azonban nem közvetlenül mutat az adatbázisra, egy táblára, egy lekérdezés eredményére, vagy egy tárolt procedúrára (stored procedure) vonatkozik. Emiatt szükségünk van még egy Table (tábla), Query (lekérdezés), vagy StoredProc (tárolt eljárás) komponensre is, ahogy azt a 16.3. ábrán is láthatjuk. Az ábra azt viszont nem mutatja, hogy adatforrásunk kapcsolódhat egy táblához vagy egy lekérdezéshez, de semmiképpen sem mind a kettőhöz egyszerre. Az viszont megengedett, hogy egy adatforráshoz több adatfüggő vezérlő is kapcsolódjon.



16.3. ábra *A DataSource komponens szerepe a különböző adatfüggő komponensek egy adatkészlethez való kapcsolása*

Amint elhelyeztünk egy táblát vagy egy lekérdezést formunkon, máris beállíthatjuk azt az adatforrás DataSet (adatkészlet) tulajdonságában. A tulajdonság megadásánál az Object Inspector megadja a választható adatkészletek listáját az aktív, és a (**File | Uses Form** paranccsal) kapcsolódó formokról. Láthatjuk tehát, hogy egy adatbázis alkalmazásnak tartalmaznia kell egy adatforrás, és valamilyen adatkészlet komponenst az formjain.

Megjegyzés

Az „adatkészlet" szó alatt mindig egy TTable, TQuery, vagy TStoredProc osztályú komponenst értek, amelyek a TBDEDataSet leszármazottjai, aminek az őse pedig a TDataSet osztály. A Delphi 3-ban létezik egy negyedik leszármazott is, a TClientDataSet osztály. Láthatjuk tehát, hogy az adatkészletek hierarchiája eléggé összetett; ne feledjük azonban, hogy a közbülső szintek új adatformák kifejlesztésére valók.

Táblák és lekérdezések

A legegyszerűbb módszer, amivel adathozzáférést biztosíthatunk a Delphiben a tábla (Table) komponens használata. Ha ezt az összetevőt alkalmazzuk, meg kell adnunk a használt adatbázis nevét a tábla DatabaseName tulajdonságánál. Itt megadhatunk egy nevet, egy álnevet (alias), vagy egy olyan könyvtár elérési útját, amely a használni kívánt táblákat tartalmazza. A választást megkönnyíti, hogy a mezőben megjelennek az elérhető adatbázisok, amelyek attól függően láthatók itt, hogy telepítettük-e őket a BDE-ben.

Meg kell adnunk a helyes táblanevet is a TableName tulajdonságnál. Ez ismét lehet egy belső használatú név, vagy egy fájl neve, ami a táblát tartalmazza. Itt is megjelennek az aktuális adatbázis választható értékei, így tehát az általános eljárás az, hogy először megadjuk a DatabaseName, majd csak utána a TableName tulajdonságot.

A táblák két másik fontos tulajdonsága a ReadOnly (csak olvasható), amely meggátolja, hogy megváltoztassuk a tartalmazott adatokat, és az Exclusive (kizárólagos), amely megtiltja, hogy hálózati környezetben egyszerre több folyamat is használja az adott táblát. Léteznek az indexekre, és a főtáblákra vonatkozó tulajdonságok is, ezeket részletesebben áttekintjük majd a fejezet végén, a MastDet példában.

A másik lehetséges adatkészlet a Delphiben a lekérdezés (Query) komponens. Egy lekérdezés általában bonyolultabb egy táblánál, mert igényel egy SQL nyelvű parancsot. Ennek ellenére könnyebben testre szabhatunk egy lekérdezést az SQL segítségével, mint egy táblát (feltéve persze, hogy ismerjük legalább a lekérdező nyelv alapjait).

A Query komponenseknek – a táblákhoz hasonlóan – van DatabaseName tulajdonságuk, de nem rendelkeznek a TableName mezővel. A táblát, amit használunk, mindig az SQL parancsban adjuk meg, amit az SQL tulajdonságban tárolunk. Ahogy később látni fogjuk, az SQL parancsnak paraméterei is lehetnek,

amelyet a Parameters mezőben adhatunk meg. A lekérdező parancsot tervezési időben a string szerkesztővel készíthetjük el, de akár futási időben is adhatunk ki ilyen parancsokat. Például írhatunk egy ilyen parancsot:

```
select * from Country
```

ahol a Country a tábla neve, a csillag (*) pedig azt jelzi, hogy a tábla minden mezőjére kíváncsiak vagyunk. Ha jól ismerjük az SQL-t, akkor nyugodtan használhatjuk gyakran, de a lekérdezések hatékonysága mindig az alkalmazott adatbázistól fog függeni. A Table komponens használata helyi táblákon gyorsabb, míg a Query hálózati környezetben ad jobb eredményeket, bár ez sem törvényszerűen van így.

Megjegyzés

Az SQL egy általános, adatbázisoknál használatos lekérdező és manipuláló nyelv. Ha nem ismerjük pontosan a használatát, a B. függelékben találhatunk ide vonatkozó információkat. A könyv megértése szempontjából azonban elég csak a legegyszerűbb parancsokat ismernünk, így nem fontos túlságosan belemerülnünk a részletekbe. A Delphi kliens/szerver változata tartalmaz egy lekérdezés-készítő eszközt, amelynek neve Visual Query Builder. Erről az eszközről a 17. fejezetben lesz részletesen szó.

Az előbbi példában a tábla és lekérdezés komponenseket egyaránt használhattuk a feladat megoldására, de ez nem mindig van így. Általában táblákat akkor használunk, mikor böngészni szeretnénk az adatok között, lekérdezést pedig akkor, amikor egy konkrét kérdésre keresünk választ, és azt szeretnénk, hogy ezt a választ az adatbázis kiszolgáló program (és nem a Delphi alkalmazás) keresse meg nekünk.

Sokszor arra is használunk lekérdezéseket, hogy kettő, vagy több táblát bizonyos szempontok szerint összekapcsoljunk, majd az eredményt úgy vizsgáljuk, mintha az is egy tábla volna. Míg a Table komponens az adatbázis egy létező tábláját mutatja, addig a lekérdezések az eredményül kapott táblákat jelenítik meg. Ez eredményezi azt, hogy az ilyen összekapcsolások, kiválasztások, vagy más számolások eredményeiben úgy böngésszünk, mintha azok is táblák lennének. Ezek a számolások – különösen az összetettek, mint az összekapcsolások – persze eltartanak egy ideig.

A harmadik típusú adatkészlet a tárolt eljárás (StoredProc), amely egy SQL kiszolgáló lokális procedúrájára mutat. Ezeket az eljárásokat lefuttathatjuk, majd az eredményt ugyancsak tábla formájában felhasználhatjuk. Ezt az eszközt kizárólag SQL kiszolgálókkal használhatjuk, éppen ezért részletes leírásuk nem található meg a könyvben.

Az adatkészlet állapota

Ha egy adatkészletet (mint például egy lekérdezés, vagy tábla) használunk, akkor a komponensnek különböző állapotai lehetnek, amik közül az éppen aktuálisat a State (állapot) tulajdonság jelöl, amely a következő értékek valamelyikét veheti fel:

- *Browse*: azt jelöli, hogy az adatkészlet normál böngésző üzemmódban van, amelyet az adatok nézegetésére, vagy a közöttük levő keresésére használunk.
- *dsEdit*: azt jelöli, hogy az adatkészlet szerkesztő üzemmódban van. Egy adatkészlet akkor kerül ilyen állapotba, amikor programunk meghívja az Edit metódust, vagy a kapcsolódó adatforrás (DataSource) AutoEdit tulajdonsága engedélyezve van, és a felhasználó egy adatfüggő vezérlőt, például egy DBGrid, vagy DBEdit komponenst használ éppen. Az adatkészlet kilép ebből az állapotból, mikor a szerkesztés befejeztével változásokat a program elküldi az adatbázis felé.
- *dsInsert*: azt jelzi, hogy egy új rekordot szúrtunk be a meglévők közé. Ez úgy történhet meg, hogy meghívjuk az Insert metódust, kiadjuk a DBNavigator komponens megfelelő parancsát, vagy a DBGrid komponens utolsó sorára mozgunk.
- *dsInactive*: az jelöli, hogy az adatkészlet le van zárva.
- *dsSetKey*: azt jelzi, hogy egy keresést készítünk elő az adatkészleten. Ez az előálló állapot a SetKey metódus, és a GotoKey, vagy a GotoNearest eljárások meghívása között (később még részletezzük ezeket a Search című példában).
- *dsCalcFields*: azt jelöli, hogy az adatkészlet rekordjai között valamilyen számolási műveletet hajtunk éppen végre, amelyet az OnCalcFields eseménykezelővel hívtunk meg. Később erre is láthatunk majd példát.
- *dsNewValue, dsOldValue, dsCurValue*: ezek az adatkészlet állapotai, amikor az ideiglenes tároló adatai frissítődnek.
- *dsFilter*: az adatkészlet akkor veszi fel ezt az állapotot, amikor egy szűrőt állítunk be, vagy az OnFilterRecord eseménykezelőt hívjuk meg.

Az egyszerűbb példákban az állapotváltás automatikusan történik, mégis fontos megértenünk ezeket, hiszen sok eljárás működése függ az adatkészlet aktuális állapotától. A fejezet későbbi példáiból (HandGri2, Navig1) többet is megtudhatunk ezekről az állapotokról, más példák pedig az adatkészletek eseményeivel foglalkoznak majd.

Más adatkezelő komponensek

A Table, Query, StoredProc, és DataSource komponensek mellett más elemek is találhatók a Data Access oldalon, a komponenspalettán:

- A *Database* komponenst átvitel-ellenőrzésre, biztonsági műveletek elvégzésére, és kapcsolat-ellenőrzésre használjuk. Általában akkor alkalmazzák, amikor egy kliens-szerver alkalmazás egy távoli adatbázishoz kapcsolódik, vagy amikor el akarják kerülni, hogy több formból kapcsolódjon az alkalmazás egy adatbázisra.
- A *Session* komponens globális rálátást biztosít az alkalmazás által kiépített adatbázis-kapcsolatokra és létező álnevekre, valamint egy eseményt biztosít az adatbázisokba való bejelentkezések testre szabásához.
- A *BatchMove* komponens segítségével olyan kötegelt műveleteket végezhetünk egy vagy több adatbázison, mint a másolás, hozzáfűzés, vagy a törlés.
- Az *UpdateSQL* elem segítségével adatfrissítő műveleteket végezhetünk SQL lekérdezések segítségével egy csak olvasható (vagyis összetett) Query komponensen. Ezt a komponenst a táblák és lekérdezések UpdateObject értékeként használjuk.
- A *Report* összetevő biztosítja a kapcsolatot a Borland ReportSmith című, jelentéskészítő alkalmazásával, amelyet néhány Delphi verzió tartalmaz.
- A *ClientDataSet* elem új a Delphi 3-ban, és a megosztott adatbáziskezelést alkalmazó programok kliens oldalán használatos.
- A *Provider* és *RemoteServer* komponensek ugyancsak újak a Delphi 3-ban, és a megosztott adatbázisok kiszolgáló oldalán használatosak. Megjegyezném, hogy az ilyen típusú alkalmazások készítése igen összetett feladat, így nem is foglalkozunk a könyvben ezzel a témával.

Ezek tehát az összetettebb adatbázis-komponensek, néhányuknak persze nem sok haszna van lokális, egyetlen számítógépből álló rendszereken. Néhányukat használjuk majd ebben, és a következő fejezetben, de nem fogunk sok időt tölteni velük.

A Delphi adatfüggő komponensei

Azt már láttuk, hogyan lehet egy adatbázishoz kapcsolni egy adatforrást, akár táblát, akár lekérdezést használva, de még mindig nem tudjuk, hogyan jeleníthetjük meg az adatokat. Ilyen célból a Delphi több komponenssel is rendelkezik, amelyek igen hasonlítanak a megszokott Windowsos vezérlőkhöz, de kapcsolatban állnak egy adatbázissal. Például a DBEdit komponens olyan, mint egy szerkesztőmező (Edit), a DBCheckBox pedig egy jelölőnégyzetre

(CheckBox) hasonlít. A komponenspaletta Data Controls (adatvezérlők) oldalán a következő elemeket találjuk:

- A *DBGrid* komponens képes egy teljes táblát megjeleníteni, lépkedhetünk a sorok és oszlopok között, görgethetjük az információkat, és szerkeszthetjük is a megjelenő rács tartalmát. Ez az összetevő a Delphi többi rácsvezérlő elemének egy kiterjesztése.
- A *DBNavigator* egy gombgyűjtemény, amely segítségével könnyedén mozoghatunk, és adhatunk ki parancsokat az adatokkal kapcsolatban. Az itt található gombok alapműveleteket végeznek, így könnyen lecserélhetjük őket saját gombjainkra.
- A *DBText* komponens olyan mező értékét jeleníti meg, amelyet nem változtathatunk meg. Ez a címkék (Label) adatbázist használó változata.
- A *DBEdit* lehetővé teszi, hogy a felhasználó megváltoztasson egy értéket az adatbázisban, egy szerkesztőmező használatával.
- A *DBMemo* segítségével nagyobb mennyiségű szöveget, vagy BLOB-ot (Binary Large OBject) adhatunk meg, vagy szerkeszthetünk át. Ez az elem a szokásos Memo komponens megfelelője.
- A *DBRichEdit* újdonság a Delphi 3-ban, segítségével formázott szöveget készíthetünk, a RichEdit komponenshez hasonlóan.
- A *DBImage* komponens segítségével egy BLOB mezőben tárolt képet jeleníthetünk meg. Ez az elem az Image összetevő kiterjesztése.
- A *DBListBox* és *DBComboBox* komponensek segítségével a felhasználó előre definiált értékek közül választhat, úgy ahogy a listapanelnél, és a kombinált listánál. Ha ezek az értékek egy adatbázis adatai közül kerülnek ki, akkor használjuk inkább helyettük a DBLookupListBox és DBLookupComboBox összetevőket, amelyek a Delphi 1 DBLookupList és DBLookupCombo komponenseit hivatottak helyettesíteni, ámbár ezek még megtalálhatók a Komponenspaletta Win 3.1 lapján.
- A *DBCheckBox*-szal egy logikai értéket jeleníthetünk meg vagy állíthatunk be, a szokásos jelölőnégyzetekhez hasonlóan.
- A *DBRadioGroup* egy rádiócsoporthoz hasonló komponens, amellyel adott választási lehetőségek közül jelölhetünk ki egyet.
- A *DBChart* a Chart komponens kiterjesztése, újdonságként jelent meg a Delphi 3-ban.
- A *DBCtrlGrid* egy többsoros rács, amely más adatfüggő vezérlőket tud tartalmazni. Az első sorhoz hozzáadott vezérlők minden rekordban megjelennek.

A felsorolt komponensek mindegyike a DataSource tulajdonságban megadott adatforráshoz kapcsolódik. Sok közülük az adatbázis egy rekordját, vagy értékét tartalmazza csupán, és hogy melyiket, azt az adatforrás megadása után választhatjuk ki a DataField (adatmező) tulajdonság segítségével. E néhány különbségtől eltekintve a bemutatott komponensek működése, tulajdonságai igen hasonlóak a nekik megfelelő standard vezérlőkéhez.

Most pedig vágjunk bele az adatbázis-kezelő alkalmazások készítésébe. Először megnézzük, hogyan lehet manuálisan elkészíteni egy ilyen programot, majd megismerkedünk az Adatbázis Form Varázslóval (Database Form Wizard).

Adatbázis-kezelő alkalmazások készítése manuálisan

Most, hogy már megismertük a Delphi különböző adatbázis-kezelő komponenseinek szerepét, készen állunk arra, hogy ilyen típusú alkalmazásokat készítsünk. A programírást néhány egyszerű példával fogjuk kezdeni. Táblákat és lekérdezéseket egyaránt fogunk alkalmazni, és használunk majd néhány adatfüggő komponenst is. Első példánk, amelyet a DBGrid összetevő segítségével készítünk el, a lehető legegyszerűbben közelíti majd meg az adatbázis-kezelés problémáját.

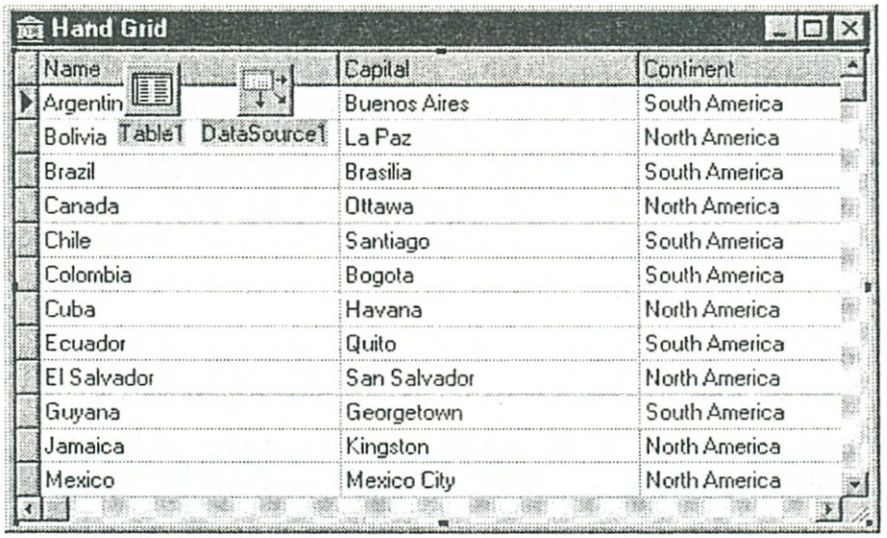
Egy adatbázis-kezelő rács

Első, HandGrid című példánk ugyanazt az amerikai országokat és fővárosokat felsoroló táblát tartalmazza, amit a 16.2. ábrán láthattunk korábban. Az egyszerűség kedvéért most ábrázoljuk az adatokat egy rács segítségével. Kezdjünk egy új formot, majd helyezzünk el rajta egy Table, egy DataSource, és egy DBGrid komponenst, majd az utóbbit igazítsuk úgy, hogy a teljes kliens területet lefedje. Ezután kapcsoljuk össze a komponenseket, vagyis állítsuk a DBGrid összetevő DataSource tulajdonságát DataSource1-re, a DataSource összetevő DataSet tulajdonságát Table1-re, végül a Table komponens DatabaseName tulajdonságát DBDEMOS-ra, a TableName tulajdonságát pedig COUNTRY.DB-re.

Ha most a táblát Active tulajdonságát igazra állítjuk, az adatok már tervezési időben megjelennek a rácson (ezt a technikát hívják live-data, vagyis élő-adat típusú tervezésnek). Ilyenkor még a rács gördítősávjait is használhatjuk, ha keresgélni akarunk az adatok, vagy a mezők között, ahogy az a 16.4. ábrán is látható.

Tipp

A 16.4. ábrán a tábla és az adatforrás komponens címkéi is látszanak. Ezeket az Environment Options **Preferences** oldalának **Show Component Captions** opciójával kapcsolhatjuk be.



16.4. ábra *A HandGrid példa formja tervezési időben*

A következő leírás a form összetevőinek, és azok tulajdonságaink szöveges leírását adja meg:

```
object Form1: TForm1
  ActiveControl = DBGrid1
  Caption = 'Hand Grid'
  object DBGrid1: TDBGrid
    Align = alClient
    DataSource = DataSource1
  end
  object Table1: TTable
    Active = True
    DatabaseName = 'DBDEMOS'
    TableName = 'COUNTRY.DB'
  end
  object DataSource1: TDataSource
    DataSet = Table1
  end
end
```

Ezután bátran futtathatjuk a programot, amely ugyanazokat az adatokat fogja nekünk mutatni, amelyeket tervezéskor már láttunk, azzal a különbséggel, hogy most már szerkeszthetjük is az értékeket Ez azért lehetséges, mert a rács Options (Beállítások) tulajdonsága tartalmazza a dgEditing kapcsolót, valamint a ReadOnly tulajdonság értéke hamis. Mivel közvetlenül az adatbázis értékeivel dolgozunk, a változtatások azonnal érvénybe lépnek.

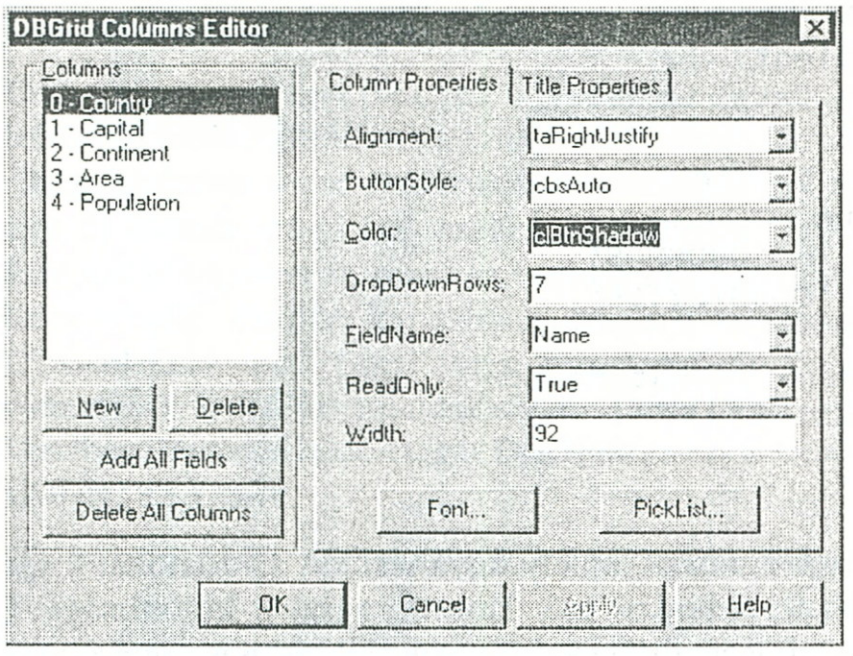
A meglévő értékek megváltoztatása mellett új rekordok beszúrására és hozzáfűzésére is van lehetőségünk. Ha egy új rekordot akarunk beszúrni, akkor nyomjuk meg az INSERT billentyűt a kívánt kurzorpozícióban. Ha csak hozzáfűzni szeretnénk egy új rekordot az adatbázishoz, akkor mozgassuk a kurzort

a rács utolsó eleme alá (vagyis menjünk az utolsó sorra, és nyomjuk meg a **lefelé nyilat**). Ha törölni szeretnénk egy rekordot, akkor nyomjuk meg a CTRL-DEL billentyű-kombinációt, majd erősítsük meg a műveletet a megjelenő párbeszédablakban. Próbálgassuk egy ideig, hogyan működik a program (esetleg készítsünk biztonsági másolatot az eredeti adatbázisról), és nézzük meg, mi lesz az eredménye, ha a rács Options tulajdonságának kapcsolóit ki-, és bekapcsoljuk.

Mi a helyzet a program kódjával? Nos, egyelőre ilyen nem létezik. A készített Pascal fájl csupán a form által használt objektumok szokásos deklarációit tartalmazza, amelyeket a Delphi automatikusan adott hozzá a programlistához. Így egyetlen programsor megírása nélkül készítettünk egy teljes értékű adatbázis alkalmazást. Ez a Delphi adatbázis programozásának kellemes oldala.

A rács testreszabása

A Delphi első változatában nem létezett semmilyen egyszerű módszer a DBGrid komponens külalakjának testreszabásához. A rendszer 32 bites változatában ehhez már létezik egy könnyen használható, mégis igen hatékony eszköz, a Columns (Oszlopok) tulajdonság. Ez a mező rendelkezik egy saját szerkesztő eszközzel is, melyet a 16.5 ábrán láthatunk.

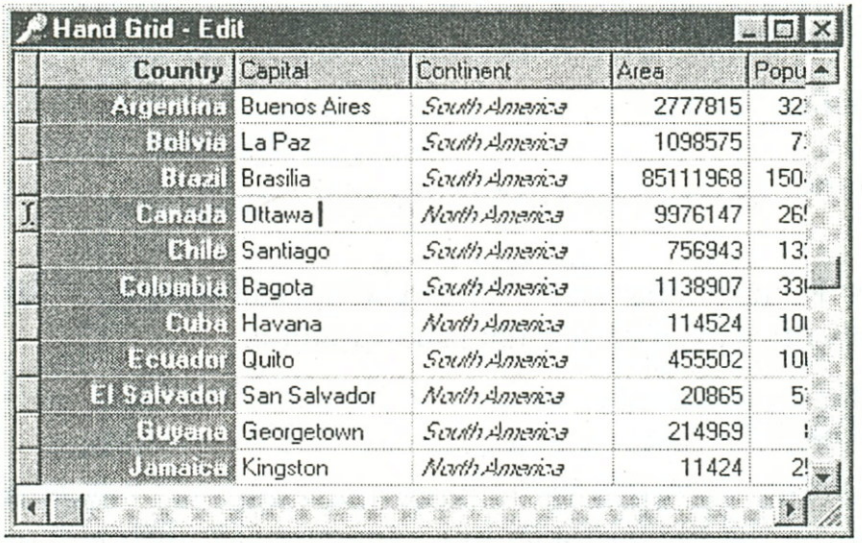


16.5. ábra *A DBGrid oszlopszerkesztője*

Itt könnyedén kiválaszthatjuk a megjelenítendő oszlopokat, és beállíthatjuk azok tulajdonságait (a **Column Properties** oldalon), a színeket, betűtípust, szélességet, igazítást, és megadhatjuk a címsor hasonló értékeit (a **Title Properties**

oldalon). A külalakon kívül azonban további, összetettebb tulajdonságok értékét is meghatározhatjuk ezen a helyen, olyanokat, mint a ButtonStyle és a DropDownRows, amelyek segítségével saját szerkesztőket, vagy legördülő listákat helyezhetünk el a rács celláiban (erre a fejezet végén látunk majd példát).

A HandGrid2 példában az előző, HandGrid című program rácsának első oszlopának feliratát, és az első és harmadik oszlop betűtípusát változtattam meg, valamint az első oszlop hátterét sötétszürkére, betűszínét pedig fehérre állítottam. Ennek eredményét a 16.6. ábra mutatja.



16.6. ábra *A HandGri2 példa futás közben*

Megjegyzés

Ha megadtuk a rács Columns tulajdonságát, akkor attól kezdve tervezési időben beállíthatjuk az oszlopok szélességét egyszerűen az egér használatával. Ezt a tulajdonságot a program futás közben is megőrzi, ha ezt beállítjuk az Options tulajdonságnál.

A tábla állapota

Sok más módon is meg lehet változtatni a rácsokat, és ezekről a módszerekről lesz is még szó a fejezetben. Most csak egy igen érdekes tulajdonságot szeretnék bemutatni egy példán keresztül. Ha megnézzük a 16.6. ábrán a form feliratát, egy újdonságra lehetünk figyelmesek: a címsorban mindig a tábla aktuális állapota látható. Vajon hogyan lehet ehhez az információhoz hozzájutni? Egyszerűen az adatforrás OnStateChange eseményének, és nem a tábla esemé-

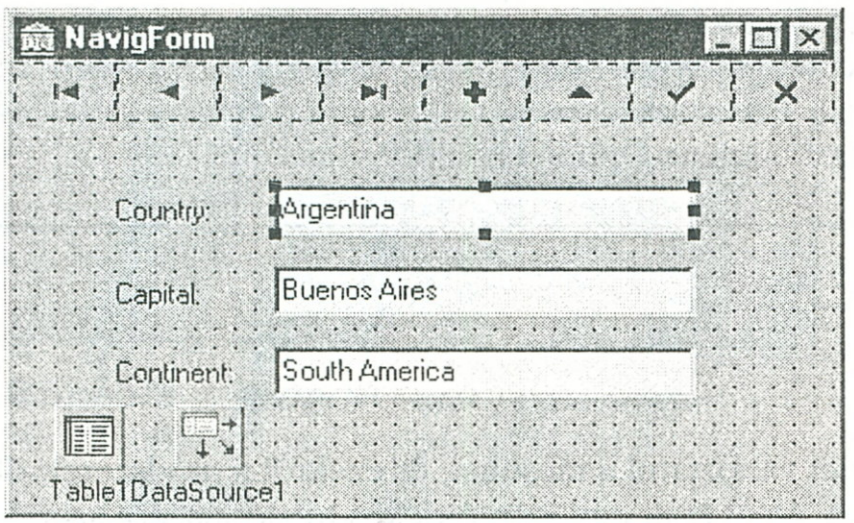
nyeinek kezelésével. Ebben a példában az eseménykezelő csupán annyit csinál, hogy egy case kifejezés segítségével megállapítja, majd kiírja a tábla állapotát:

```
procedure TForm1.DataSource1StateChange(Sender:TObject);
var
  Title: string;
begin
  case Table1.State of
    dsBrowse: Title := 'Browse';
    dsEdit: Title := 'Edit';
    dsInsert: Title := 'Insert';
  else
    Title := 'Other state';
  end;
  Caption := 'Hand Grid - ' + Title;
end;
```

A programrészlet a Table komponens azon három állapotát dolgozza fel, amely a felhasználó által kiváltott események hatására előállhat.

DBEdit vezérlők használata

A HandGri2 példa jól működik, de most próbálkozzunk meg más vezérlők használatával is, és csupán a minket érdeklő információkat jelenítsük meg programunkkal, ne az egész adatbázist. A következő, Navig1 című példa a tábla és az adatforrás mellett DBEdit komponenseket, néhány egyszerű feliratot, és egy eddig nem alkalmazott, DBNavigator nevű összetevőt fog használni. A 16.7. ábra mutatja a Navig1 program tervezés alatti képét.



16.7. ábra *A Windows Másol és Beilleszt parancsával könnyedén másolhatunk táblákat, vagy vezérlőket egyik formról a másikra, tulajdonságaikkal együtt. Az éppen tárgyalt esetben ez a módszer igen hasznos lehet*

Ismét össze kell kapcsolnunk az adatfüggő vezérlőket az adatforrással, úgy, hogy beállítjuk a DataSource tulajdonságukat. Meg kell még adnunk a három szerkesztőmező DataField (adatmező) tulajdonságát (Name, Capital és Continent a példában). Ha már hozzákapcsoltuk a táblához és a szerkesztőmezőkhöz az adatforrást, akkor egyszerűen válasszuk ki a DataField megfelelő értékét az Object Inspector által felajánlottak közül. Ha a kapcsolat létrejött, és a tábla Active tulajdonságát igazra állítottuk, az első rekord értékei megjelennek a szerkesztőmezőkben (lásd 16.7. ábra).

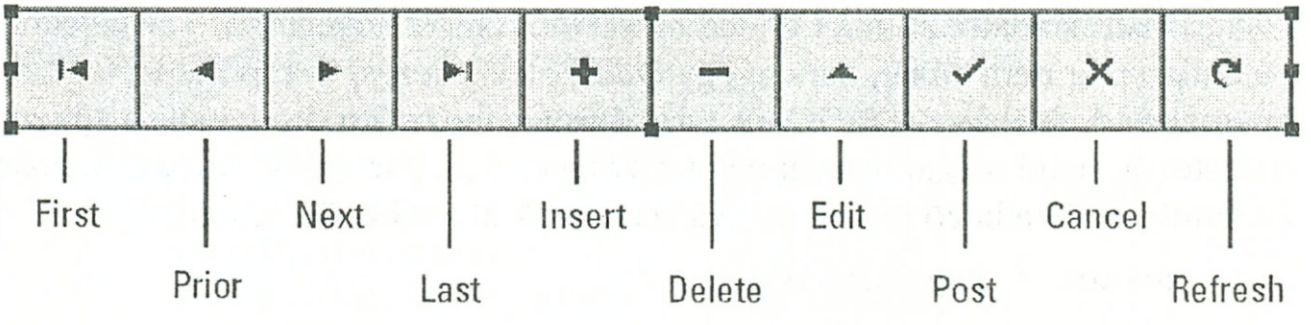
A következő lépésként kikapcsoljuk a DBNavigator néhány gombját. Ezt a VisibleButtons (látható gombok) halmaz megfelelő elemeinek eltávolításával érhetjük el. Az elemek jelentését a 16.1. táblázat mutatja

16.1. táblázat

A DBNavigator vezérlő gombjainak jelentése

Gomb	Jelentés
nbFirst	Ugrás az első rekordra
nbPrior	Ugrás az előző rekordra
nbNext	Ugrás a következő rekordra
nbLast	Ugrás az utolsó rekordra
nbInsert	Új üres rekord beszúrása az aktuális pozíción
nbDelete	Az adott rekord törlése
nbEdit	Az aktuális rekord szerkesztésének engedélyezése
nbPost	Az adatok változtatásainak mentése
nbCancel	Az adatok változtatásainak elvetése

A vezérlő gombjait és a hozzájuk tartozó műveleteket a 16.8. ábrán láthatjuk. Sajnos a gombok szimbólumai nem túl beszédesek, de szerencsére a komponens automatikusan megjelenő lebegő tippekkel rendelkezik, amik akkor válnak láthatóvá, mikor a felhasználó az egérmutatót a gombok fölé húzza.



16.8. ábra *A DBNavigator komponens gombjai*

Tipp

A ShowHint tulajdonság kikapcsolásával letilthatjuk a navigátor megjelenő tippjeit, vagy még jobb, ha a Hints stringlista segítségével saját tippeket készíthetünk. Ez jól jöhet, ha nem angol nyelvű programot készítünk. A beillesztett stringek a gombok sorrendjében kell hogy álljanak: az első az első gombra vonatkozik, a második a második gombra, és így tovább.

A Navig1 programban én is csak néhányat használtam a DBNavigator gombjai közül, kikapcsoltam a törlő és a frissítő funkciókat. A komponenst a form tetejéhez igazítottam, és a Flat tulajdonságát igazra állítottam (ahogy azt már a 12. fejezetben megbeszéltük). Futtassuk le, és próbáljuk ki, hogy működik-e; azt is észrevehetjük, hogy az ablak felirata itt is a tábla állapotának megfelelően változik. Ez azért van így, mert a HandGri2 példa adatforrását használtam a program elkészítéséhez.

Vegyük észre, hogy a program futásának kezdetekor, és használat közben az első, illetve az utolsó rekordra ugráskor, a navigátor megfelelő gombjai automatikusan kikapcsolódnak. Ha azonban úgy érjük el az adatbázis elejét (vagy végét), hogy egyenként lépkedünk a rekordok között, akkor a program csak akkor kapcsolja ki a megfelelő gombokat, ha megpróbáljuk átlépni a határokat. Ez azért van így, mert a komponens csak ekkor ismeri fel, hogy az adott irányban nincs több rekord. Ugyanez az esemény játszódik le, amikor a rács függőleges gördítősávját használjuk. A többi gombbal nincs ilyen gond: azok megfelelően működnek a szerkesztő üzemmódba való be- és kilépéskor egyaránt.

Lekérdezések használata

Az első két példában, ahol egy táblát használtunk a program elkészítéséhez, végigjárhattuk adatbázisunk minden rekordját. A most következő, Navig2 című példában már nem táblát, hanem egy Query (lekérdezés) komponenst fogunk használni. A szokásos DBDEMO adatbázishoz hozzákapcsolhatjuk a Query összetevőt, majd megadhatunk egy tetszőleges SQL parancsot. Ahogy azt már láthattuk, a következő paranccsal választhatjuk ki a tábla összes mezőjét:

```
select * from Country
```

Miután megadtuk beírtuk ezt a sort, és aktivizáltuk a lekérdezést az Active tulajdonság igazra állításával, az első rekord mezőinek értékei megjelennek a szerkesztőmezőkben. Természetesen ennek az a feltétele, hogy a beírt SQL parancs helyes legyen; ellenkező esetben a Delphi egy hibaüzenetet küld számunkra, és nem futtatja le a lekérdezést.

Megjegyzés

Ha meg akarjuk változtatni egy lekérdezés SQL parancsát, akkor annak Active tulajdonságát előbb hamisra kell állítanunk. Ezután változtathatunk a parancson, majd hogy az életbe lépjen, újra aktiválni kell a lekérdezést. Másik, hasonló megoldás lehet még az Open és a Close eljárások meghívása.

Persze ez a példa nem túl érdekes, hiszen ugyanazt a hatást értük el, mintha egy táblát használtunk volna. A következő program azonban bemutatja a lekérdezések előnyeit a táblával szemben: futási időben fogunk négy különböző kérdést intézni az adatbázishoz, négy különböző rádiógomb segítségével.

Az első gombbal a már említett lekérdezést valósíthatjuk meg, és ez a gomb van bejelölve a program indulásakor is. A második és a harmadik gombbal előre adott, hogy miket válogathatunk ki: azokat a rekordokat, amelyek észak- vagy dél-amerikai országok adatait tartalmazzák. Ezt úgy érhetjük el, hogy a `where` szót illesztjük a parancs végére, ahogy azt hamarosan látni fogjuk. A negyedik gomb lehetővé teszi, hogy a felhasználó tetszőleges szöveget gépeljen be a `where` után, meghatározva ezzel a lekérdezés eredményét (a `where` részletes leírását megtalálhatjuk a B függelékben).

A felhasználó által begépelt parancsok persze veszélyesek, hiszen semmi sem biztosítja azok helyességét. Szerencsére a Delphi rendelkezik egy olyan eszközzel, amivel kezelni tudunk ilyen helyzeteket: ez az eszköz a kivételkezelés. A fejezetben először elérkezett az idő, hogy hozzáadjunk néhány sort programunkhoz, hiszen csak így érhetjük el, hogy a rádiógombok közötti kapcsolgatással a lekérdezés SQL parancsa is megváltozzon. Nem szabad elfelejtenünk, hogy minden alkalommal, amikor végrehajtjuk ezt a műveletet, a lekérdezésen egy Close (Bezár) majd egy Open (Megnyit) műveletet kell végrehajtanunk, esetleg az Active tulajdonságot hamisra, majd újra igazra kell állítanunk. Íme az első rádiógombhoz tartozó kódrészlet:

```
procedure TNavigForm.RadioButton1Click(Sender: TObject);
begin
  Query1.Close;
  Query1.Sql.Clear;
  Query1.Sql.Add('select * from Country');
  Query1.Open;
end;
```

Vegyük észre, hogy az SQL tulajdonság nem string, hanem TStrings típusú. Ennek segítségével hosszú lekérdezéseket is tudunk használni, hiszen az ilyen

típusú szövegek igen hosszúak lehetnek, és a lekérdezés egyes részeit a program különböző pontjain adhatjuk meg. A részeket később egyszerűen összefűzzük. A példában én egy hagyományosabb módszert alkalmaztam, hiszen csak egy egysoros lekérdezést használok. A második és harmadik rádiógomb kódja ugyanaz, mind a kettő saját Caption tulajdonságát használja az SQL parancs felépítéséhez.

```
procedure TNavigForm.RadioButton2Click(Sender: TObject);
begin
  Query1.Close;
  Query1.Sql.Clear;
  Query1.Sql.Add ('select * from Country');
  Query1.Sql.Add ('where Continent = "' +
    (Sender as TRadioButton).Caption + '"');
  Query1.Open;
end;
```

Ez a programrészlet két string összeadásával készít el egy SQL parancsot. Az SQL utasításokra szabadon alkalmazhatjuk a megszokott string műveleteket, használhattam volna egy vagy akár három stringet is.

Tipp

A fenti példa SQL parancsában dupla idézőjellel (") jelöltem a stringet, noha ezt szimplával (') is csinálhattam volna. Ez azonban a Pascal stringképzési szabályai miatt nagyon elbonyolította volna a programsorokat, így a duplánál maradtam.

Az utolsó rádiógombhoz tartozó programrészlet egyszerűbb, hiszen itt csak hozzá kell fűznünk az alapértelmezés szerinti parancshoz a szerkesztőmező tartalmát:

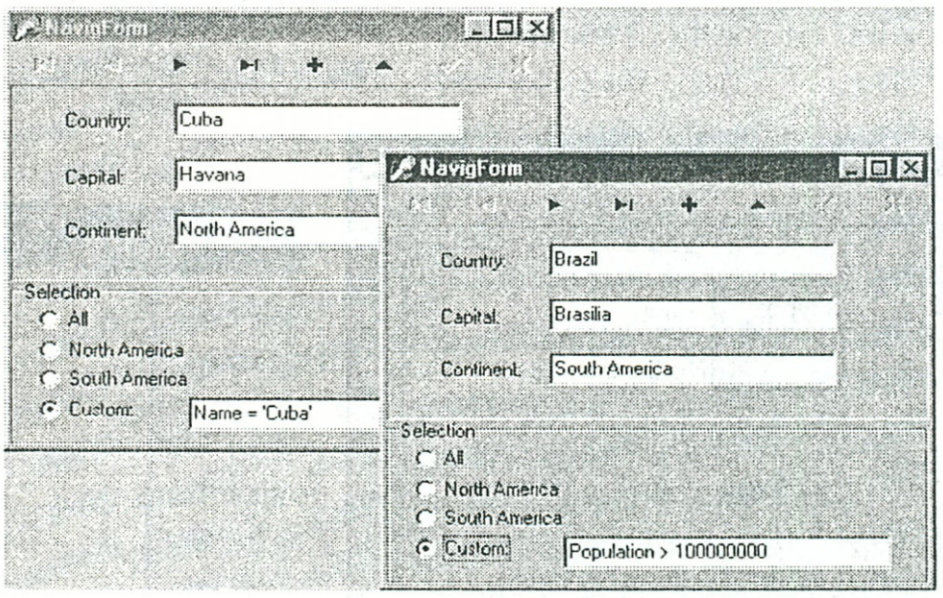
```
procedure TNavigForm.RadioButton4Click(Sender: TObject);
begin
  Query1.Close;
  if (Edit1.Text <> '') then
  begin
    Query1.Sql.Clear;
    Query1.Sql.Add ('select * from Country');
    Query1.Sql.Add ('where ' + Edit1.Text);
  end;
  Query1.Open;
end;
```

Ezt a részt a program mindig végrehajtja, amikor a szerkesztőmező nem üres (feltételezve, hogy a beírt SQL parancs helyes, hiszen ezt a program nem ellenőrzi). Továbbfejleszthetjük a programunkat úgy is, hogy ezt az utolsó rádiógombot ne tudjuk kiválasztani, ha a szerkesztőmező üres. Ezt az ellenőrzést a szerkesztőmező OnChange eseményénél végezzük:

```
procedure TNavigForm.Edit1Change(Sender: TObject);
begin
  RadioButton4.Enabled := Edit1.Text <> '';
end;
```

A gombok kiválasztásakor az aktuális lekérdezés eredménye rögtön megjelenik a képernyőn. Fontos, hogy a navigátor mindig az utolsó lekérdezés eredményén hajtja végre a kiadott parancsokat, így helyesen mutatja azt, hogy Kanada az első az észak-amerikai államok névsorrendjében. A Custom nevű szerkesztőmezőt többféleképpen is használhatjuk: a 16.9. ábra két példát is mutat erre. Az egyikben az adatbázis egyetlen kiválasztott rekordját láthatjuk, a másikban a népesség alapján végzünk egy szelekciót. Ha valami értelmetlen dolgot írunk a mezőbe, vagy szintaktikai hibát vétünk, a program hibaüzenettel leáll.

Az utolsó fontos dolog a példával kapcsolatban az, hogy alapértelmezés szerint nem szerkeszthetjük a lekérdezés eredményét. Ha ezt szeretnénk elérni, akkor a Query komponens RequestLive tulajdonságát igazra kell állítanunk. Azonban ez is csak néhány fontos feltétel teljesülése esetén teszi lehetővé, hogy a lekérdezés eredményét módosítsuk; általában az egyetlen táblára vonatkozó lekérdezések eredményeit szerkeszthetővé tehetjük, míg a komplex lekérdezésekét, amelyek több táblát is összekapcsolnak, nem.



16.9. ábra *A Navig2 példa két példánya. Az egyikben egy konkrét értéket keresünk, a másikban pedig a nagy népességgel rendelkező országokat válogatjuk ki*

Megjegyzés

SQL parancsok használata helyett a táblánál is beállíthatjuk a keresett rekordok határértékét. Ennek a határnak egy indexen kell alapulnia. Egy másik megoldás lehet a szűrések használata. Egyszerűen állítsuk a tábla Filtered tulajdonságát igazra, és adjunk meg egy feltételt a Filter mezőben. A program ekkor az OnFilterRecord esemény meghívásával fogja kiválogatni a megjelenítendő rekordokat.

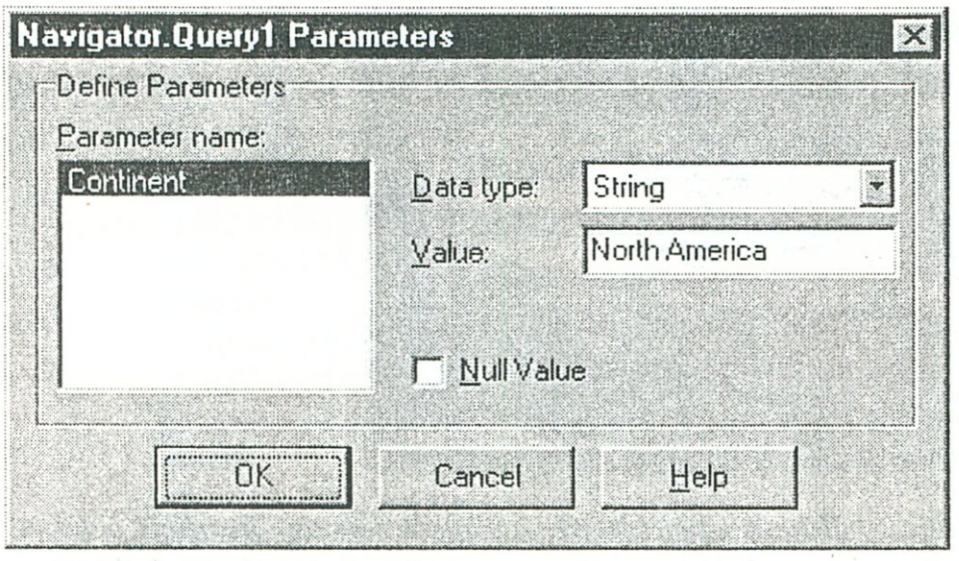
Paraméterezett lekérdezések

Az utolsó példa bemutatta, milyenek a különböző paraméterrel rendelkező lekérdezések. Ahelyett azonban, hogy minden alkalommal újra felépítsük SQL parancsainkat, lehetőségünk van arra, hogy csupán annak paramétereit változtassuk meg.

Ha úgy döntünk, hogy az észak- és dél-amerikai államok közül akarunk választani, akkor megadhatjuk a következő lekérdezést is:

```
select * from Country where Continent = :Continent
```

Ebben a parancsban a :Continent rész a paraméter. A lekérdezés parancs tulajdonságánál egy speciális szerkesztő segítségével beállíthatjuk a paraméterek típusát és kezdeti értékét. Ezt a szerkesztőt az Object Inspectoron vagy a form gyorsmenüjének **Define Parameters** (paraméterek megadása) parancsán keresztül érhetjük el, a Query komponens kiválasztása után. A megnyitott szerkesztőben láthatjuk a lekérdezésben megadott paramétereket, a hozzájuk tartozó adattípussal és kezdeti értékkel (16.10. ábra).



16.10. ábra *A Query komponens paraméter-szerkesztője*

Ezzel az új eszközzel az előző példát leegyszerűsíthetjük. A form komponensei közül csak két rádiógomb marad meg, a program legfontosabb része pedig ezen gombok OnClick eseményének kezelése lesz.

```
procedure TNavigForm.RadioButton2Click(Sender: TObject);
begin
  Query1.Close;
  Query1.Params[0].AsString :=
    (Sender as TRadioButton).Caption;
  Query1.Open;
end;
```

A fenti eljárás bemásolja a rádiógomb feliratát a lekérdezés első paraméterébe. Erre a paraméterre a Params tömb nullával sorszámozott elemeként hivatkozhatunk, és string típusú értéket adhatunk át neki. Az AsString tulajdonság egy általános jellemzője az adatbázis komponensek paraméter és mező tömbjeinek. Ahogy később látjuk majd, ezen tömbök elemei különböző adattípusúak lehetnek, így fontos, hogy figyeljünk, hogy a megfelelő típusokat használjuk.

Hozzáadhatunk még egy eljárást a programhoz, ha ki szeretnénk használni a paraméterezett lekérdezések előnyeit. Figyeljünk a paraméterek felhasználásakor, hisz egy kis optimalizálással a lekérdezéseket felgyorsíthatjuk. Ezt az optimalizálást előkészítésnek nevezzük. Egyszerűen hívjuk meg a Prepare (előkészít) eljárást, mielőtt programunk megnyitná a lekérdezést. Példánkban, melynek címe Navig3, a lekérdezés Active tulajdonságát hamisra kell állítanunk még tervezéskor, majd a form OnCreate kezelőjéhez a következőket kell hozzáadnunk:

```
procedure TNavigator.FormCreate(Sender: TObject);
begin
  Query1.Prepare;
  Query1.Open;
end;
```

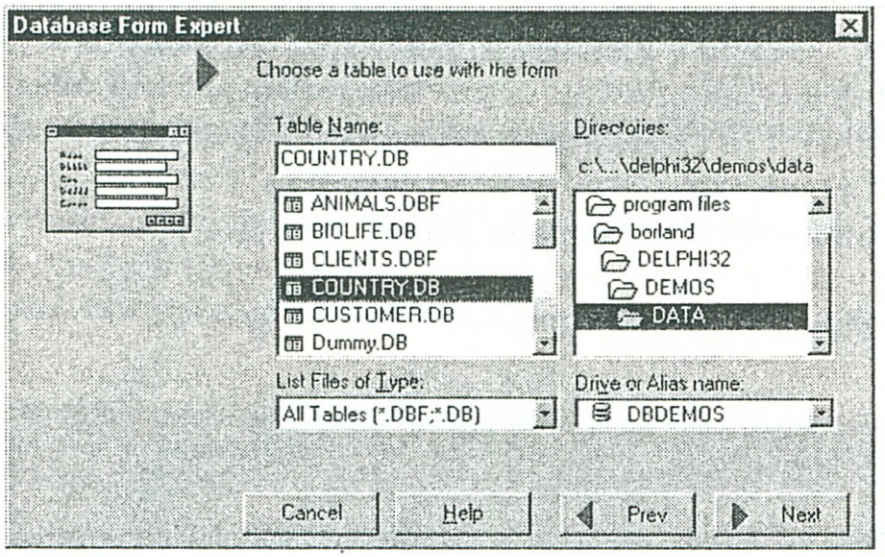
Megjegyzés

Az előkészített paraméteres lekérdezéseknek igen nagy jelentőségük van nagy táblák és komplex lekérdezések használatakor. Az ilyen lekérdezések optimalizálása érdekében sok adatbázis-kezelő ideiglenes indexeket hoz létre. Az előkészített lekérdezés elmenti az ilyen indexeket, így az nem kell minden művelet alkalmával létrehozni, amivel sok időt spórolhatunk meg. Elképzelhető, hogy az általunk használt SQL kiszolgáló automatikusan elvégzi ezt az előkészítést, ilyenkor a műveletnek nem lesz további hatása.

A Database Form varázsló használata

Most már tudjuk, hogyan kell egyszerű alkalmazásokat készíteni az adatbázis komponensek formra helyezésével és egymáshoz kapcsolásával. Természetesen ez a művelet időt vesz igénybe, ezért fejlesztőkörnyezetünk tartalmaz egy Database Form varázslót (ezt a kiegészítést a Delphi előző verzióiban Database Form Expert-nek nevezték). Erről az eszközről esett már szó a harmadik fejezetben, ahol a Delphi objektumtárolóját boncolgattuk (Object Repository). Itt az idő, hogy használjuk is, és megértsük az általa készített programot.

Csak gyakorlásképpen megpróbálhatjuk újra felépíteni a fejezet első HandGrid című példáját, a varázsló segítségével. Nevezzük el ezt a programot ExpGrid-nek. Kezdjünk egy új, üres projektet, és indítsuk el a Database Form varázslót. Válasszuk az egyszerű táblán alapuló formot, válasszuk a COUNTRY.DB táblát, a DBDEMOS adatbázisból, jelöljük ki az összes mezőt, és válasszunk egy rácsot. Az előző lépések egyikét a 16.11. ábrán láthatjuk. Most már elkészíthetjük a formot, eltávolíthatjuk az eredeti üres formot a projektből és lefordíthatjuk a programot. A fájlok megfelelő névvel történő mentése után a programot már futtathatjuk is. A készített alkalmazás hasonló lesz a korábbihoz, de az elkészítésre fordított idő lényegesen kevesebb volt.



16.11. ábra *Az országokat ábrázoló rács elkészítésének második lépése*

A varázsló még ennél is gyorsabban működik, ha formunk egy egyszerű rács helyett címkéken és szerkesztőmezőkön alapul. Ebben az esetben a komponensek formra helyezése, elrendezése és a kapcsolódó műveletek elkészítése igen sok ideig tartana. A varázsló segítségével ez a hosszadalmas művelet egyetlen kattintással elvégezhető, csupán választanunk kell a vízszintes és függőleges elrendezés között. Ha függőleges elrendezést választottuk, a címkék elhelyezését is megválaszthatjuk. A fejezetben erre a későbbiekben láthatunk egy példát is.

Mielőtt újabb alkalmazásokat készítenénk, szánjunk egy percet a varázsló által készített kód tanulmányozására. Az ExpGrid formjának hat komponense van, ahogy azt láthatjuk is, ha megnyitjuk a CD-n található projekt fájt, vagy gyorsan készítünk egy új alkalmazást.

A példa egy adatokhoz kapcsolt táblát egy adatforrást egy navigátort (két beágyazott panellel) és egy rácsot tartalmaz. A paneleket a navigátor jobbraigazítására használjuk úgy, hogy emellett ne kerüljön zavaróan közel a kerethez. A példa kódja igen egyszerű: az egyetlen előre elkészített eljárás a form OnCreate eseményére vonatkozik, és a tábla Open metódusának meghívásával annak megnyitására használjuk.

Igaz ugyan, hogy a Database Form varázsló nem készít hosszú programokat, de segítségével bonyolult formokat és összetett táblákat vagy lekérdezéseket építhetünk. Sajnos a varázslót nem lehet testreszabni, mivel a fejlesztőkörnyezet nem tartalmazza az ehhez szükséges forrásfájlokat.

Az adatmezők elérése

Van még egy-két fontos technikai részlet, amelyekkel meg kell ismerkednünk, mielőtt bonyolultabb és látványosabb programok készítésére adjuk a fejünk. Egészen mostanáig a tábla minden mezőjét felhasználtuk programjainkban. Biztosan észrevettük, hogy a Database Form varázsló lehetőséget nyújt arra, hogy válasszunk az alkalmazni kívánt mezők közül. Hogyan tudjuk akkor megoldani ezt a problémát, ha már kész alkalmazással dolgozunk? Hogyan adhatunk táblánkhoz olyan új elemeket, mint például a számított mező? Ezen problémák boncolgatásakor más gondok is felmerülhetnek; egyáltalán hogyan lehet az aktuális rekord értékeit programunkból elérni, és hogyan változtathatjuk meg őket?

Ezen kérdésekre a válasz a mezők természetében rejlik. A Field (mező) komponensek (amelyek a TField osztály, vagy alosztályainak példányai) a Delphi adatbázis-kezelésének alapjául szolgálnak. Az adatfüggő vezérlők közvetlenül az adatbázis mezőire vonatkozó Field objektumokhoz kapcsolódnak.

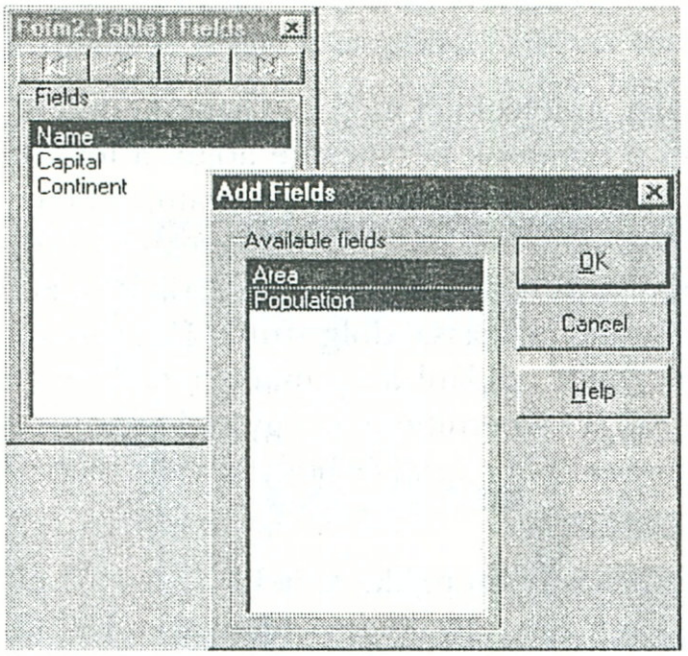
Az eddig felépített alkalmazásokban a Delphi automatikusan elkészítette futásidő alatt a TField komponenseket. Ez minden olyan esetben megtörténik, mikor adatkészletet nyitunk meg. Ezeket a mezőket a táblák és lekérdezések a Field tulajdonságban tárolják, amely egy mezőkből álló tömböt tartalmaz. A tömb elemeit számozással vagy nevükre való hivatkozással érhetjük el (a FieldByName metódussal):

```
Table1.Fields[0].AsString
Table1.FieldByName('Name').AsString
```

Másik megoldásként a mezőket már tervezés alatt létrehozhatjuk a Fields Editor (mezőszerkesztő) segítségével. Ha ezt választjuk, akkor a mezők tulajdonságait

már tervezés alatt beállíthatjuk. Ezek a tulajdonságok a mezőkhöz kapcsolódó adatfüggő vezérlők viselkedését és megjelenését módosítják. Az új mező elkészítése után azonnal megjelenik az Object Inspectorban a többi komponens mellett.

Próbáljuk most ezt ki úgy, hogy elindítjuk a mezőszerkesztőt az utolsó példa táblájával. Válasszuk ki a formról a táblát az egér jobb gombjával jelenítsük meg annak helyi menüjét, majd válasszuk a **Fields Editor** parancsot. Ekkor egy üres mezőszerkesztő jelenik meg a képernyőn. Következő lépésként a szerkesztő helyi menüjét kell használjuk. A legegyszerűbb elvégezhető művelet az **Add** (hozzáad) parancs használata, amellyel a mezők listájához újabb elemeket adhatunk hozzá. Az **Add Fields** (mezők hozzáadása párbeszédablak, amely az elérhető mezők listáját tartalmazza, a 16.12. ábrán látható, ezek azok az elemek, amelyek még nem szerepelnek a szerkesztő mezőket tartalmazó listáján.



16.12. ábra *A mezőszerkesztő Add Fields ablaka*

A **Define** (meghatározás) parancs segítségével új számított mezőket, keresőmezőket vagy módosított típusú mezőket hozhatunk létre. Az ekkor megjelenő párbeszédablakban megadhatunk egy jellemző mezőnevet, amely szóközöket is tartalmazhat. A Delphi készít egy belső nevet is, ami a mező komponens elnevezése lesz, amelyet mi később megváltoztathatunk. Következő lépésként a mező adattípusát kell kiválasztanunk. Ha egy számított vagy keresőmezőről van szó, akkor egyszerűen válasszuk ki a megfelelő opciót. A következő részben megnézzük, hogyan készíthetünk számított mezőket a fejezet végén pedig azt, hogy hogyan lehet keresőmezőt létrehozni.

Megjegyzés

A TField komponens rendelkezik egy Name (név), és egy FieldName (mezőnév) tulajdonsággal. A Name a megszokott módon használandó, a FieldName segítségével az adatbázis egy oszlopát, vagy a számított mező nevét állíthatjuk be. Itt hosszabb leírást is adhatunk, és szóközöket is használhatunk. A FieldName tulajdonság értékét alapértelmezés szerint megkapja a DisplayLabel tulajdonság, amit aztán tetszés szerint átírhatunk. A mezőnevet többek között a FieldByName metódus is felhasználja keresésre.

Minden létrehozott mező megtalálható a mezőszerkesztőben, s elérhető az adatfüggő vezérlők számára. Az eredeti adatbázis egy táblájának mezője nincs rajta ezen a listán, akkor ezt nem is használhatják ezek a vezérlők. A mezőszerkesztő használatakor az elérhető mezők deklarációit a Delphi hozzáadja a form osztályához, úgy, ahogy a menütervező ad TMenuItem komponenseket a formhoz. A TField osztály leszármazottjai nem láthatóak a formon, de kiválaszthatjuk őket az Object Inspector segítségével. Természetesen programszövegben is hivatkozhatunk rájuk, megváltoztathatjuk a futás alatt is tulajdonságaikat, lekérhetjük vagy beállíthatjuk értékeiket.

A mezőszerkesztő segítségével a mezők egymáshoz viszonyított sorrendje is megváltoztatható, ami rácsok készítésénél fontos, hiszen ez a komponens a mezőket azok sorrendje alapján jeleníti meg.

Tipp

Egy kellemes tulajdonsága a mezőszerkesztőnek, hogy ha az egérrel egy mezőt a formra húzunk, akkor a Delphi automatikusan elkészít egy, a mezőre vonatkozó vezérlőt. Ezzel nagyon meggyorsíthatjuk munkánkat, magam is sokat használom, ajánlom, hogy ha még nem próbáltuk, készítsünk el egy alkalmazást ezzel a módszerrel.

A mezőosztályok hierarchiája

Mielőtt megnézünk egy példát, mélyedjünk bele egy kicsit a TField osztályok használatába. Soha ne becsüljük alá ennek a komponensnek a jelentősségét. Igaz, hogy sokszor csak a színfalak között használjuk, de nagyon fontos szerepet játszik az adatbázis-kezelő alkalmazások működésében. Ahogy már említettem, még ha nem is hozunk létre ilyen objektumokat, akkor is elérhetjük a

tábla mezőit a Field tömb vagy a FieldByName metódus használatával. Mind a két eszköz egy TField típusú objektumot ad vissza, így szükségünk lehet az as operátor használatára, hogy a megfelelő típusúvá alakítsuk őket (pl. TFloatField vagy TDateField).

A FieldAcc példa az ExpGrid egy kiterjesztése. A program három gombbal és egy eszköztárral rendelkezik, és különböző futási idejű mezőtulajdonságokat használ. Az első gomb a rács népességet nyilvántartó oszlopának formátumát változtatja meg. Ehhez a DisplayFormat tulajdonságot használjuk, amely a TFloatField osztályhoz tartozik. Az ide tartozó programrészlet a következő:

```
procedure TForm2.SpeedButton1Click(Sender: TObject);
begin
  (Table1.FieldByName ('Population') as
    TFloatField).DisplayFormat := '###,###,###';
end;
```

Ha adatbevitelre és megjelenítésre vonatkozó tulajdonságot állítunk be, akkor a változás a tábla minden rekordjában érvénybe lép. Ha egy mező értékére vonatkozó változtatásokat végzünk, akkor az mindig csak az aktuális rekordra vonatkozik. Például megjeleníthetjük az aktuális ország népességét egy üzenetablakban a következő sorok segítségével:

```
procedure TForm2.SpeedButton2Click(Sender: TObject);
begin
  ShowMessage (Table1.FieldByName('Name').AsString +
    ': ' + Table1.FieldByName('Population').AsString);
end;
```

Mikor egy mező értékét használjuk, az As tulajdonságok egy sorozatát kell alkalmaznunk ha a tartalmazott értékre vagyunk kíváncsiak.

```
AsBoolean: Boolean;
AsDateTime: TDateTime;
AsFloat: Double;
AsInteger: LongInt;
AsString: string;
```

Ezeket a tulajdonságokat használhatjuk egy mező értékének kiolvasására vagy módosítására. Változtatni csak akkor tudunk, ha az adatkészlet szerkesztő módban van. Az előbb említett módszer alternatívája lehet az, ha a mezők Value tulajdonságát használjuk, ami Variant típusú. Annak ellenére, hogy ez néha hasznos számunkra, a Variant típus használata jelentősen lelassíthatja programunkat, ahogy azt a negyedik fejezetben említettem.

A TField komponens többi tulajdonságának nagy része (pl. Alignment, DisplayLabel, DisplayWidth, Visible) a mező megjelenését határozza meg, és ezeket főleg különböző adatfüggő vezérlők használják. A FieldAcc példában a harmadik gomb minden mező Alignment tulajdonságán változtat:

```
procedure TForm2.SpeedButton3Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to Table1.FieldCount - 1 do
    Table1.Fields[I].Alignment := taCenter;
end;
```

Ez főleg a DBGrid és az eszköztárhoz hozzáadott DEdit vezérlő megjelenésén változtat. Az új megjelenítési formát a 16.13. ábra mutatja.

A VCL sok különböző mezőosztály típust tartalmaz. A Delphi ezeket futásidő alatt és a mezőszerkesztő használatakor automatikusan alkalmazza. A 16.2. táblázat a TField osztály alosztályát mutatja be.

Name	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2777815	32,300,003
Bolivia	La Paz	South America	1098575	7,300,000
Brazil	Brasilia	South America	85111968	150,400,000
Canada	Ottawa	North America	9976147	26,500,000
Chile	Santiago	South America	756943	13,200,000
Colombia	Bagota	South America	1138907	33,000,000
Cuba	Havana	North America	114524	10,600,000
Ecuador	Quito	South America	455502	10,600,000
El Salvador	San Salvador	North America	20865	5,300,000
Guyana	Georgetown	South America	214969	800,000

16.13. ábra *A FieldAcc példa futásának eredménye a Center és a Format gombok megnyomása után*

16.2. táblázat

A TField osztály leszármazottjai

Alosztály	Ősosztály	Meghatározás
TStringField	TField	Fix hosszúságú szöveges adat (maximum 255 karakter hosszú lehet, adatbázistípustól függően.
TNumericField	TField	Általában nem használjuk közvetlenül. Ez minden numerikus mezőosztály őse.
TIntegerField	TNumericField	Hosszú egész (32 bites) számok.
TSmallIntField	TIntegerField	Egész, 16 bites számok.
TWordField	TIntegerField	Előjel nélküli 16 bites egész számok.

16.2. táblázat

A TField osztály leszármazottjai (folytatás)

Alosztály	Ősosztály	Meghatározás
TAutoIncField	TIntegerField	Speciális egészek, amelyeket a rendszer a mezők indexelésére használ, értékük automatikusan növekszik.
TFloatField	TNumericField	8 bájtos lebegőpontos számok.
TCurrencyField	TFloatField	Pénznem. Mérete megegyezik a valós számok méretével.
TBCDField	TNumericField	Olyan valós számok, amelyek fix mennyiségű tizedesjegyet tartalmaznak.
TBooleanField	TField	Logikai érték.
TDateTimeField	TField	Dátum és idő.
TDateField	TDateTimeField	Dátum.
TTimeField	TDateTimeField	Idő.
TBinaryField	TField	Általában nem használjuk közvetlenül. Ez a két következő osztály őse.
TBytesField	TBinaryField	Méretkorlátozás nélküli tetszőleges adat.
TVarBytesField	TBytesField	64K méretű tetszőleges adat.
TBlobField	TField	Méretkorlátozás nélküli bináris adat (a BLOB a Binary Large OBject rövidítése).
TMemoField	TBlobField	Tetszőleges hosszúságú szöveg.
TGraphicField	TBlobField	Tetszőleges méretű grafikus objektum.

Megjegyzés

Az elérhető mezők típusa nagyban függ a használt adatbázistól. Az InterBase például nem támogatja a BCD-t, így ilyen kiszolgálókkal nem használhatunk BCDField mezőket. A lebegőpontos számok mérete és a velük végzett műveletek pontossága szintén függ az adatbázis típusától.

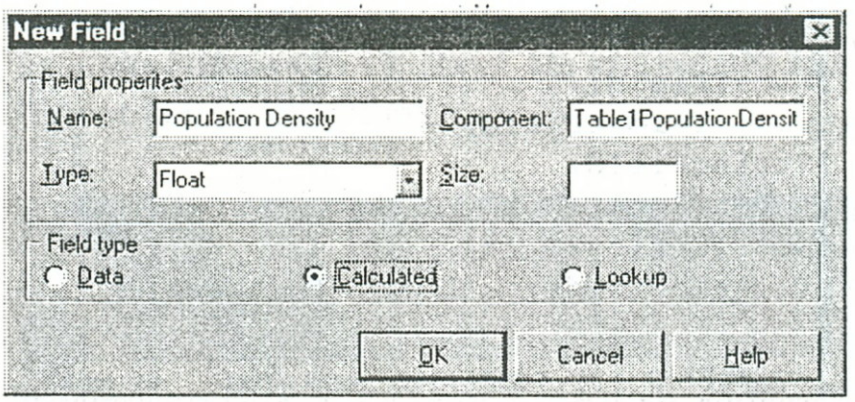
Számított mező készítése

Most, hogy megismerkedtünk a TField objektummal, és példákat láttunk futásidejű használatára, itt az idő, hogy elkészítsünk egy alkalmazást a mezőszerkesztő segítségével. Ismét az ExpGrid példából fogunk kiindulni, amelyhez hozzáadunk majd egy számított mezőt. Rendelkezésünkre áll minden ország népessége és területe, így kiszámíthatjuk a népsűrűségi adatokat.

Az új példa elkészítéséhez, melynek címe Calc, helyezzünk el egy táblát a formon, majd nyissuk meg a mezőszerkesztőt a form gyorsmenüjének segítségével. A szerkesztőben az Add paranccsal válasszunk ki néhány mezőt (a példa készítésekor én mindet kiválasztottam)., majd a Define paranccsal adjunk nevet és adattípust (TFloatField) az új számított mezőnek, úgy, ahogy azt a 16.14. ábrán láthatjuk.

Természetesen azt is meg kell mondanunk, milyen műveletekkel szeretnénk kiszámítani az új mező értékeit. Ezt a tábla OnCalcFields eseményénél adjuk meg, a következő sorokkal:

```
procedure TForm2.Table1CalcFields(DataSet: TDataSet);
begin
  Table1PopulationDensity.Value :=
    Table1Population.Value / Table1Area.Value;
end;
```



16.14. ábra *A Calc példa számított mezőjének definíciója*

Most azt hihetnénk, hogy minden rendben, ám ez nincs így. Ha egy új rekordot adunk a meglévőkhöz, és a területet véletlenül nullára állítjuk, akkor a program az osztás elvégzésekor egy kivételt fog kiváltani, és le fog állni. Az egyik megoldás lehet az. hogy egyszerűen kezeljük a kivételt, és nullára állítjuk a végeredményt:

```
try
    Table1PopulationDensity.Value :=
      Table1Population.Value / Table1Area.Value;
  except
    on Exception do
      Table1PopulationDensity.Value := 0;
  end;
```

Létezik azért ennél jobb megoldás is. Le tudjuk ellenőrizni, hogy tartalmaz-e a mező megadott értéket, és hogy az nem nulla-e. Jobb nem használni kivételeket, ha meg tudjuk előzni az esetleges hibákat.

```
if not Table1Area.IsNull and
       (Table1Area.Value <> 0) then
    Table1PopulationDensity.Value :=
      Table1Population.Value / Table1Area.Value
  else
       Table1PopulationDensity.Value := 0;
```

A fenti Table1CalcFields metódus néhány mezőt közvetlenül használ. Ez azért lehetséges, mert a mezőszerkesztőt használtam, és ez automatikusan elkészítette az ide vonatkozó meződeklarációkat, ahogy azt a form interface deklarációjában láthatjuk.

```
type
  TCalcForm = class(TForm)
    Table1: TTable;
    Table1PopulationDensity: TFloatField;
    Table1Area: TFloatField;
    Table1Population: TFloatField;
    Table1Name: TStringField;
    Table1Capital: TStringField;
    Table1Continent: TStringField;
    procedure Table1CalcFields(DataSet: TDataset);
    ...
```

Akár hozzáadunk, akár eltávolítunk mezőket a mezőszerkesztő segítségével, műveletünk hatását azonnal látni fogjuk a rácson. Természetesen a számított mezők értékeit nem mutatja a Delphi tervezéskor, ezek csak futásidőben érhetők el, hiszen lefordított Pascal kód futásának eredményei. A mezők néhány komponensét arra is használhatjuk, hogy a rács külalakján változtassunk. Például ha azt szeretnénk, hogy számainkban az ezreseket vesszők válasszák el, akkor az Object Inspectorral néhány mező DisplayFormat tulajdonságát állítsuk a következőre: „###,###,###". A változtatás azonnal életbe lép.

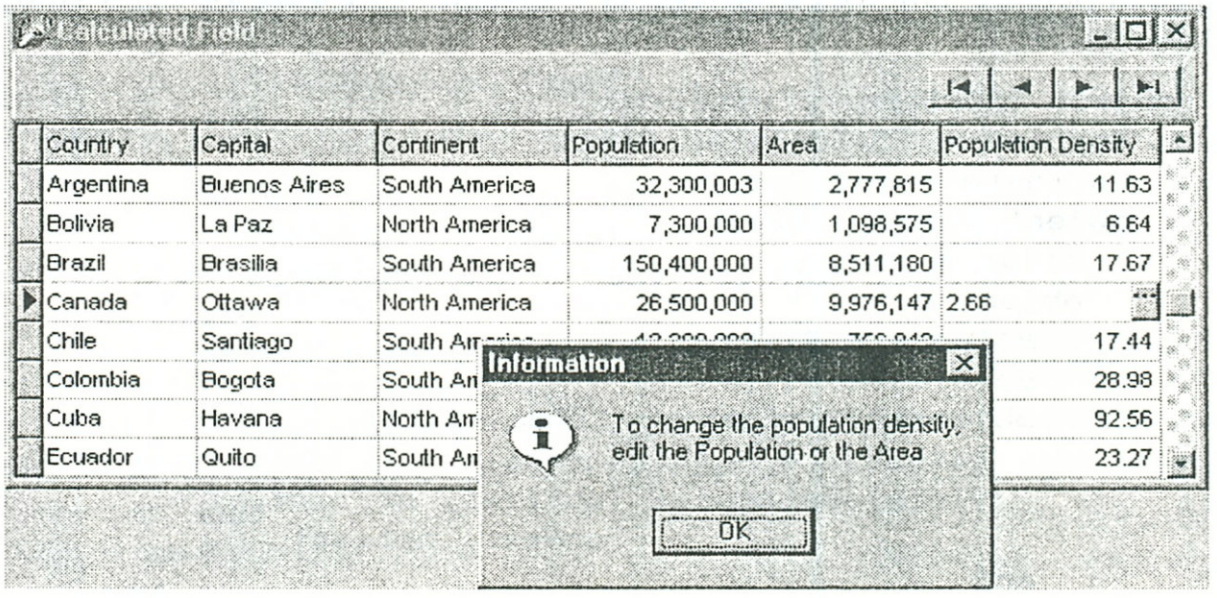
Megjegyzés

Az előbb említett megjelenítési formátum a Windows International Settings értékeit használja. A szimbólumok aktuális értékre való cseréjekor a Delphi a Windows nemzetközi beállításait veszi figyelembe.

A tábla összetevőin a mezőkön végzett munka után tovább alakítottam a DBGrid komponenst annak Columns tulajdonság szerkesztőjével. A Population Density (népsűrűség) oszlopot csak olvashatóra állítottam, valamint a ButtonStyle tulajdonságnál cbEllipsis értéket adtam meg. Ha változtatni próbálunk ezen az értéken, egy kis gomb jelenik meg a cellában, a gomb megnyomása a DBGrid OnEditButtonClick eseményét hívja meg.

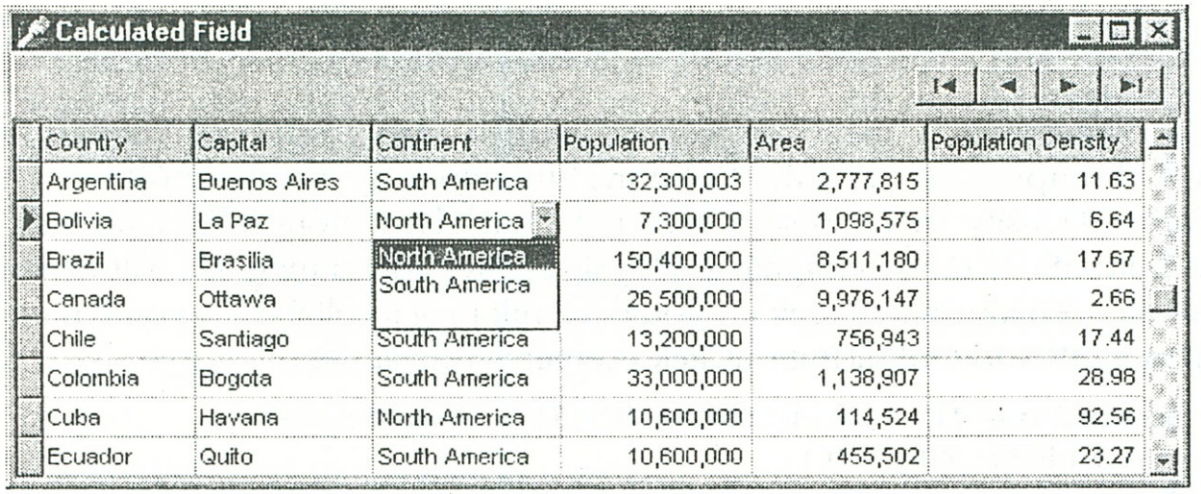
```
procedure TCalcForm.DBGrid1EditButtonClick(
  Sender: TObject);
begin
  MessageDlg ('To change the population density,'#13 +
    'edit the Population or the Area',
    mtInformation, [mbOK], 0);
end;
```

Valójában itt nem egy szerkesztőt indítunk el, csupán egy egyszerű üzenetablakot, amely tájékoztatja a felhasználót arról, hogy hogyan lehet megváltoztatni az adott értéket (16.15. ábra). Ha szerkesztőt szeretnénk készíteni, egy újabb formot kell létrehoznunk.



16.15. ábra *A Calc példa futásának eredménye. A Population Density oszlopban megtalálható a létrehozott gomb, az üzenet pedig a párbeszédablakban olvasható*

A második testreszabott oszlop a Continent (kontinens). Ennél a mezőnél csak két választási lehetőséget szeretnék felkínálni, ezért kattintsunk a PickList gombra és gépeljük be a használni kívánt opciók listáját, majd állítsuk be az értékek számát a DropDownRows tulajdonságba. További változtatás nem szükséges.



16.16. ábra *A rács PickList tulajdonságával beállított legördülő lista*

Ha változtatni próbáljuk a mező értékét, megjelenik a legördülő lista. A Columns tulajdonság végleges értékeit mindig a megfelelő szerkesztőben vagy a form szöveges leírásában találhatjuk meg. Ez a tulajdonság és a mezőobjektumok határozzák meg a rács külalakját és kulcsszerepet játszanak az alábbi példa DFM fájljában. Felhívom a figyelmet a form varázsló által készített két beágyazott panelre, amelyek segítségével a navigátort a form jobb felső sarkában jelenítjük meg.

```
object CalcForm: TCalcForm
  Caption = 'Calculated Field'
  OnCreate = FormCreate
  object Panel1: TPanel
    Align = alTop
    object Panel2: TPanel
      Align = alRight
      BevelOuter = bvNone
      object DBNavigator: TDBNavigator
        DataSource = DataSource1
        VisibleButtons = [nbFirst, nbPrior,
                          nbNext, nbLast]
      end
    end
  end
  object DBGrid1: TDBGrid
```

```
    Align = alClient
    Columns = <
      item
        FieldName = 'Name'
      end
      item
        DropDownRows = 3
        FieldName = 'Continent'
        PickList.Strings = (
          'North America'
          'South America')
      end
      ...
      item
        ButtonStyle = cbsEllipsis
        FieldName = 'Population Density'
        ReadOnly = True
      end>
    DataSource = DataSource1
    OnEditButtonClick = DBGrid1EditButtonClick
  end
  object DataSource1: TDataSource
    DataSet = Table1
  end
  object Table1: TTable
    Active = True
    OnCalcFields = Table1CalcFields
    DatabaseName = 'DBDEMOS'
    TableName = 'country.db'
    object Table1Name: TStringField
      DisplayLabel = 'Country'
      FieldName = 'Name'
    end
    ...
    object Table1Area: TFloatField
      FieldName = 'Area'
      DisplayFormat = '###,###,###'
    end
    object Table1PopulationDensity: TFloatField
      Calculated = True
      FieldName = 'Population Density'
      DisplayFormat = '###.##'
    end
  end
end
```

Mezők használata a tábla változtatásához

A TField komponens arra is használható, hogy futási időben lekérdezzünk, vagy megváltoztassunk adatokat a táblában. Egyelőre csak korlátozott használatát láttuk a közvetlen adatelérésnek, az előző példában két mezőt használtunk arra, hogy kiszámoljuk a harmadik értékét. Most néhány egyszerű alkalmazást készítünk el, melyek a mezők használatával lehetővé teszik, hogy keressünk a táblában, műveleteket végezzünk az értékekkel, vagy információkat szerezzünk az adatbázisról. Természetesen sok más művelet is végezhető a mezőkkel, de ötletadónak remélhetőleg ezek is megteszik majd.

Rekordok keresése a táblában

A következő két példához egy új formra lesz szükségünk, amelyet ezúttal az EMPLOYEE.DB táblához kapcsolunk, amely szintén a Delphi példaadatbázisai közül való. A több szerkesztőmezőt tartalmazó form elkészítéséhez használhatjuk a Database Form varázslót, vagy elkészíthetjük azt elemenként felépítve.

Ha a varázslót használjuk, akkor először meg kell szabadulnunk az eszköztárban levő DBNavigator komponenstől. Az alapértelmezés szerinti navigációs komponensek helyett hat új gombot rakunk fel a formra. A két szélső (amit << és >> felirattal jelölünk) segítségével az első illetve az utolsó rekordra ugorhatunk, a két középsővel (amit < és > felirat) egyenként lépkedhetünk a rekordok között, a maradék kettővel („+5", „-5") pedig ötösével ugorhatunk előre illetve hátra.

Mivel már sok gomb van a formon, (és később még néhány továbbit is létrehozunk) mindegyiknek beszédes nevet adtam (pl. SpeedButtonFirst, SpeedButtonNext, stb.). Minden gombhoz tartozik egy kódrészlet, melyet annak OnClick eseményénél adtunk meg. Lényegét tekintve ez a hat metódus mindegyike a táblában való mozgást segíti elő. Itt látható a hat eljáráshívás, melyeket a forráskódból másoltam ki, a gombok sorrendjének megfelelően:

```
Table1.First;
Table1.MoveBy (-5);
Table1.Prior;
Table1.Next;
Table1.MoveBy (5);
Table1.Last;
```

A táblában való mozgás egyszerű dolog, a programot az bonyolítja el, hogy az eszköztár gombjainak Enabled tulajdonságait is kezelni kell. Azokat a gombokat, amelyek az előrefelé mozgást segítik elő, ki kell kapcsolnunk, amikor a tábla végére érünk. Ennek pont az ellenkezőjét kell tennünk a tábla legelején.

Azt, hogy elértük-e már a tábla elejét vagy végét, a BOF (Beginning of File – a fájl eleje) és az EOF (End Of File – a fájl vége) tulajdonságokkal ellenőrizhetjük

le. A gond az, hogy többféleképpen is elérhetünk a tábla végére, és többféleképpen is elhagyhatjuk azt. Íme az egyik azon eljárások közül, melyek ki-, illetve bekapcsolják azokat.

```
procedure TSearchForm.DisableNextButtons;
begin
  SpeedButtonLast.Enabled := False;
  SpeedButtonNext.Enabled := False;
  SpeedButtonMoveOn.Enabled := False;
end;
```

Ezt az eljárást akkor hívjuk meg, mikor a fájl végére érünk. Ehhez hasonlatos eljárást készítettem a Next gombhoz is:

```
procedure TSearchForm.SpeedButtonNextClick(Sender:
                                                TObject);
begin
  Table1.Next;
  EnablePriorButtons;
  if Table1.EOF then
    DisableNextButtons;
end;
```

Ebben a pár sorban minden alkalommal bekapcsoljuk az első három gombot (nem jelent nagy időveszteséget akkor sem, ha a gomb már engedélyezett volt), és kikapcsoljuk az utolsó hármat, ha az EOF tulajdonság értéke igaz. A program többi részét megtalálhatjuk a mellékelt CD-n.

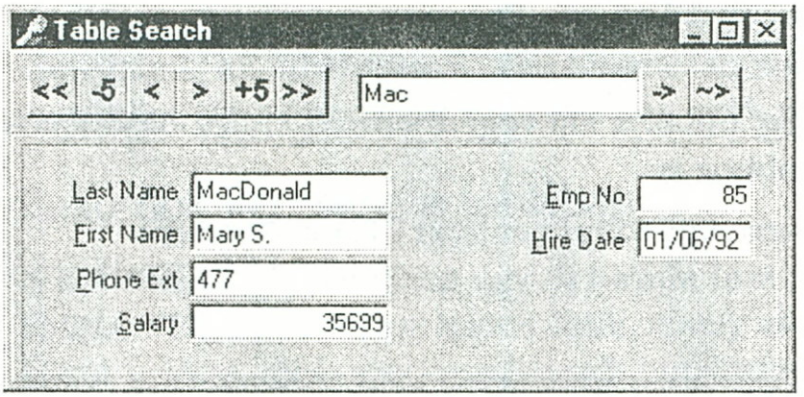
Ha már működnek a gombjaink, továbbfejleszthetjük példánkat keresési funkciót készítve. Szeretnénk elérni, hogy egy szerkesztőmezőben megadott értékhez tudjunk ugorni.

Mielőtt folytatnánk a példa megbeszélését, vessünk egy pillantást a 16.17. ábrán található formra, amely a végső állapotot tükrözi. Különösen felhívom a figyelmet az adatfüggő szerkesztőmezőt tartalmazó panelekre. Ezeket a vezérlőket a Database Form varázslóval készítettem. Ezek tökéletesen megfelelnek, szabadon átméretezhetjük a formot. Ha a form nem elég nagy, hogy megjelenítse az összes mezőt, automatikusan görgetősávok jelennek meg rajta.

A keresést két új gomb, és a hozzájuk tartozó szerkesztőmező segítségével végezhetjük. Az első gombbal pontos egyezést, a másodikkal pedig a legközelebbi egyezést kereshetjük meg. Mindkét esetben a megadott szöveget az Employee tábla LastName mezőjéhez hasonlítjuk. A keresést a Table komponens különböző metódusaival végezhetjük el, pl. GotoKey, FindKey, GotoNearest, FindNearest.

Ha index nélküli mezőben keresünk, akkor általában a Locate metódust használhatjuk. Ezzel a módszerrel mindig az optimálisabb utat választhatjuk a feladat végrehajtására; indexelt táblákban az index segítségével, index nélküliekben pedig lineárisan keres.

Ha a keresőmetódusok első csoportját szeretnénk használni, akkor be kell állítanunk a tábla IndexFieldNames tulajdonságát a helyes értékre (ebben az esetben közvetlenül kiválaszthatjuk a LastName és FirstName értékeket). Ha nincsen index, akkor létre kell hozzunk egy másodlagos indexet a Database Desktop vagy a Database Explorer használatával. Az utóbbit a már létező indexek megtekintésére is használhatjuk.



16.17. ábra *A Search példa közelítő keresése*

A kereső metódusok

Az index pontos beállítása után megkezdhetjük magát a keresést. A legegyszerűbb módszer erre a FindNearest metódus használata, amely egy közelítő keresést végez, a FindKey eljárással pedig pontos egyezéseket kereshetünk.

```
procedure TSearchForm.SpeedButtonGoNearClick(Sender:
                                                TObject);
begin
  Table1.FindNearest ([EditName.Text]);
  EnableAllButtons;
end;

procedure TSearchForm.SpeedButtonGotoClick(Sender:
TObject);
begin
  if not Table1.FindKey ([EditName.Text]) then
    MessageDlg ('Name not found', mtError, [mbOk], 0)
  else
    EnableAllButtons;
end;
```

Mindkét keresőmódszer konstansok egy tömbjét használja paraméterként. Minden ilyen paraméter egy indexelt mezőre vonatkozik. Esetünkben csupán az első mező értékét adjuk át, így a keresés a többit nem veszi figyelembe.

Goto metódusok

Amint azt láthattuk, a FindNearest és a FindKey metódusok használata igen egyszerű. Hogy jobban megértsük működésüket, ismerkedjünk meg a GotoNearest és a GotoKey eljárásokkal. A két utóbbi eljárás igen közel áll a konkrét alacsonyszintű BDE hívásokhoz. E kettő közül az egyszerűbb a GotoNearest, amellyel szintén közelítő keresést végezhetünk.

```
procedure TSearchForm.SpeedButtonGoNearClick(Sender:
                                                TObject);
begin
  Table1.SetKey;
  Table1.FieldByName('LastName').AsString :=
      EditName.Text;
  Table1.GotoNearest;
  EnableAllButtons;
end;
```

Ahogy azt a programban is láthatjuk, a keresés három lépésben zajlik:

- A táblát keresőállapotba állítjuk.
- Megadjuk a mezők célértékeit. A példában a LastName mező értékét használtam fel. A FieldByName metódust azért használtam közvetlen mezőelérés helyett (Table1.Fields[1]), mert így a program olvashatóbb.
- Végül elindítjuk a keresést, és a kívánt pozícióban lévő rekordot választjuk ki aktuálisnak.

A példa működését a 16.17. ábrán láthatjuk. A programban találhatunk egy sort, mely segítségével a navigációs eszköztár gombjait elérhetővé tesszük.

A másik módszer a pontos egyezést kereső algoritmus használata. A kód két sorban különbözik az előzőtől:

```
procedure TSearchForm.SpeedButtonGotoClick(Sender:
                                                TObject);
begin
  Table1.SetKey;
  Table1.FieldByName('LastName').AsString :=
      EditName.Text;
  Table1.KeyFieldCount := 1;
  if not Table1.GotoKey then
    MessageDlg ('Name not found', mtError, [mbOK], 0)
  else
    EnableAllButtons;
end;
```

Ahogy már említettem, a programnak helyesen beállított indexre van szüksége. Felhívom a figyelmet a KeyFieldCount tulajdonság értékére, amely mutatja,

hogy az indexet felépítő két mező közül csak az elsőt használjuk. A második különbség pedig az eljárások sikeressége: a GotoNearest pontos egyezés hiányában is eredményes, míg a másik programban a GotoKey metódus, amennyiben nem talál egyezést, hibaértékkel tér vissza.

A FindKey eljárás ugyanúgy működik, mint a GotoKey, a különbség csupán annyi, hogy míg a FindKey-t egyszerűbb használni, addig a GotoKey hatékonyabb hibakezelést tesz lehetővé.

A Locate metódus

Ha táblánk illetve a használt mező nem rendelkezik indexekkel, akkor a fenti technikák nem alkalmazhatóak. Ilyen esetekben használjuk a Locate metódust. Ez a módszer jól használható általános problémák megoldására, hiszen nem függ az index létezésétől, milyenségétől. Elég intelligens hozzá, hogy amennyiben létezik egy számára megfelelő index, azt használja, amennyiben nem talál ilyet, egy lassabb, de indexet nem igénylő eljárást használ a kereséshez.

A Locate használata is igen egyszerű: Első paraméterként a mező nevét, másodikként pedig a keresett értékeket várja. Íme egy példa használatára, mely a Search programból származik:

```
if not Table1.Locate ('LastName', EditName.Text, [])
then
  MessageDlg ('Name not found', mtError, [mbOk], 0)
else
  EnableAllButtons;
```

A mellékelt CD-n található Search példában a program három változatát is megtalálhatjuk.

Egy tábla oszlopának szummázása

Eddigi példáinkban megtekinthettük egy adatbázis adatait, és kézzel szerkeszthettük azokat, illetve új rekordokat vehettünk fel. Most azt nézzük meg, hogyan is lehet megváltoztatni programunkkal egy tábla adatait. Példának tekintsük a következőt: Egy programot kell készítenünk egy vállalat vezetője számára, aki a következőt kéri tőlünk: A dolgozókat tartalmazó táblában nem csak egyenként szeretné a fizetéseket megváltoztatni, hanem például tudjon mindenkinek egyszerre 10% fizetésemelést adni, és könnyen megtudhassa, hogy a vállalat összesen mennyi fizetést ad dolgozóinak.

Készítsük el ezt a programot, nevezzük Total-nak. Induljunk ki az előző példából, és fejlesszük azt tovább a kívánt irányba. Eszköztárunkba felveszünk két új gombot, és egy SpinEdit (egy léptetőgombbal ellátott mező).

A Fields Editor használatával megnyitottam a használt táblát, eltávolítottam a Table1Salary mezőt, amely TFloatField-ként szerepelt, majd a New Field

parancs segítségével egy új mezőt hoztam létre ugyanezen a néven, de TCurrencyField típussal. Ez nem számított mező, tulajdonságai megegyeznek a TFloatField tulajdonságaival, csupán a megjelenített adatokat pénznem formátumban mutatja.

Nézzük most meg az új program kódját. Mindenek előtt a szummázó gombbal foglalkozunk, melyen két dollárjel ($$) található, ahogy azt a 16.18. ábra is mutatja. E gomb segítségével számíthatjuk ki a fizetések összegét, változtathatunk ezek értékein, majd újra szummázhatunk. A programrész a Salary mező értékeinek végigolvasásán alapul.

```
procedure TSearchForm.SpeedButtonTotalClick(
  Sender: TObject);
var
  Total: Real;
begin
  Total := 0;
  Table1.First;
  while not Table1.EOF do
  begin
    Total := Total + Table1Salary.Value;
    Table1.Next;
  end;
  MessageDlg ('Sum of new salaries is ' +
    Format ('%m', [Total]), mtInformation, [mbOK], 0);
end;
```

A program működik ugyan, mint ahogy azt a 16.18. ábrán is láthatjuk, de van vele néhány gond is. Az első az, hogy a kurzor az utolsó rekordra kerül, így annak eredeti pozíciója elvész. Ezt a problémát az aktuális kurzorpozíció eltárolásával oldhatjuk meg. A tárolást egy úgynevezett tábla könyvjelző (TBookmark) segítségével végezzük, amely egy különleges, csak erre a célra használható változó. A program elején létrehozunk egy ilyen típusú változót, és kezdeti értékként megadjuk neki a kurzor aktuális pozícióját.

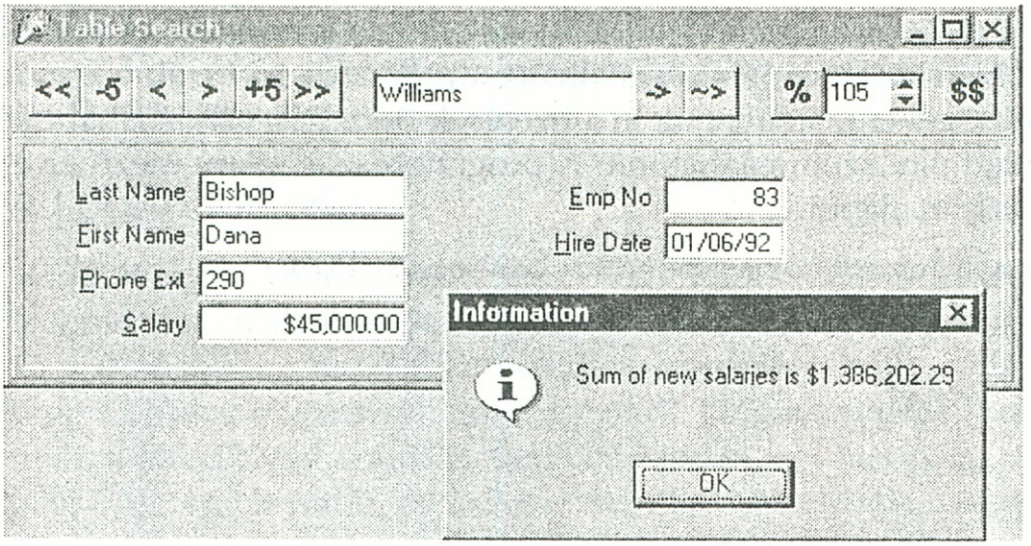
```
var
  Bookmark: TBookmark;
begin
  Bookmark := Table1.GetBookmark;
```

A SpeedButtonTotalClick eljárás végén visszaállíthatjuk az eredeti kurzorpozíciót, és törölhetjük a könyvjelzőt a következő két sorral.

```
Table1.GotoBookmark (Bookmark);
Table1.FreeBookmark (Bookmark);
```

A másik gond programunkkal az, hogy a számoláshoz szükséges lépések elvégzésekor a képernyőn lévő vezérlők tartalma is változik, követve a műveleteket. Hogy ezt elkerüljük, a számolás idejére ki kell kapcsolnunk a vezérlőket,

a tábla DisableControls metódusával. A számolás végeztével az EnableControls-szal kapcsolhatjuk vissza, természetesen a rekordmutató eredeti helyének visszaállítása után. Ha ezt nem tesszük meg, összezavarjuk a felhasználót.



16.18. ábra *A Total című program végeredménye, az alkalmazottak fizetésének összegével*

Megjegyzés

Az adatfüggő vezérlők kikapcsolása nem csak szebbé, de gyorsabbá is teszi a programot, hiszen a megjelenítésre szánt idő sokkal több, mint a számolás ideje. Próbáljuk lefuttatni programunkat a DisableControls és EnableControls eljárások kihagyásával, és látni fogjuk a különbséget.

Végül szembe kell néznünk a tábla adatainak olvasásakor felmerülő hibákkal is, különösen, ha programunk hálózaton fut. Ilyen hibák esetén programunk egy kivételt (exception) indít el, a vezérlők kikapcsolt állapotúak maradnak, és a program nem folytathatja normális működését. A megoldás egy try-finally blokk használata. Ha még biztosabbra akarunk menni, akkor alkalmazzunk két egymásba ágyazott ilyen blokkot. Az említett változtatások után programunk így néz ki:

```
procedure TSearchForm.SpeedButtonTotalClick(
  Sender: TObject);
var
  Bookmark: TBookmark;
  Total: Real;
```

```
begin
  Bookmark := Table1.GetBookmark;
  try
    Table1.DisableControls;
    Total := 0;
    try
      Table1.First;
      while not Table1.EOF do
      begin
        Total := Total + Table1Salary.Value;
        Table1.Next;
      end;
    finally
        Table1.EnableControls;
  end
  finally
    Table1.GotoBookmark (Bookmark);
    Table1.FreeBookmark (Bookmark);
  end;
  MessageDlg ('Sum of new salaries is ' +
    Format ('%m', [Total]), mtInformation, [mbOK], 0);
end;
```

A példában végigmentünk egy tábla tartalmán, hogy megmutassam, hogyan is lehet a rekordokon egyesével végigmenni, de ugyanezt az eredményt sokkal gyorsabban is elérhettük volna egy SQL parancs segítségével (lásd a B. függelék példáját). Az SQL parancsok segítségével végzett lekérdezések és változtatások általában is gyorsabbak, mintha a fenti programhoz hasonlóan végezzük azokat. SQL kiszolgáló használata esetén a hatékonyságbeli különbség még tovább nőhet.

A tábla egy oszlopának szerkesztése

Az új gomb OnClick eseménye által kiváltott eljárás hasonló az előző programrészlethez. Ez is végigmegy a táblán, és kiszámolja a fizetések összegét. Annak ellenére, hogy csak két utasításban találunk eltérést a fenti és e programok között, működésük mégis lényegesen különbözik. A fizetés növelésekor egy adatot változtatunk meg a táblában. Az ehhez tartozó sorok a következők.

```
while not Table1.EOF do
begin
  Table1.Edit;
  Table1Salary.Value := Round (Table1Salary.Value *
    SpinEdit1.Value) / 100;
  Total := Total + Table1Salary.Value;
  Table1.Next;
end;
```

Az első utasítás szerkesztő módba állítja a táblát, így a változtatások azonnal megtörténnek, a második kiszámolja az új fizetést az eredeti értéket megszorozva az SpinEdit komponensben tárolt értékkel (alapértelmezés szerint 105), majd elosztva százzal. Ez egy 5%-os béremelést eredményez, az értékeket a program a legközelebbi egészre kerekíti. A programmal most már tetszés szerint változtathatjuk, akár meg is duplázhatjuk a fizetéseket egyetlen gombnyomással. Ne feledjük, hogy ezzel a művelettel töröljük az eredeti értékeket a táblából. Az adatokat „visszaállíthatjuk", ha az elvégzett számolási művelet inverzét végeztetjük el, ám vigyázzunk, mivel a kerekítés miatt nem biztos, hogy pontos eredményt kapunk.

Amennyiben az általunk használt adatbázis támogatja tranzakciók használatát, akkor egy tranzakciót kezdhetünk a ciklus futása előtt, és a ciklus végeztével lezárhatjuk vagy törölhetjük (elvethetjük) azt. A tranzakció törlésekor minden, annak kezdése után kiadott parancs elvész, a rendszer nem végzi el. Ez garantálja, hogy minden fizetés egyszerre változzon meg. Természetesen, ha a változásokat elvégeztük (a tranzakciót lezártuk), a műveleteket már nem vonhatjuk vissza.

Az adatbázis különböző tábláinak használata

Eddigi példáinkban a használni kívánt táblát mindig tervezési időben választottuk ki. De mit tehetünk, ha nem tudjuk előre, hogy futáskor melyik táblával kell műveleteinket végeznünk. Először azt gondolhatnánk, hogy ilyen programot nem is lehet írni ebben a rendszerben, de ez nem így van. Természetesen egyszerűbb, ha mindent tervezési időben állítunk be, a paraméterek futásidejű állításához hosszabb programokat is kell írnunk. Pontosan ezt tesszük a következő, Tables nevű példánkban, amely bemutatja, hogyan érhetjük el a BDE számára látható adatbázisokat és azok tábláit futás alatt.

Az adatbázis és a tábla kiválasztása futásidőben

A példaprogram első része igen egyszerű. Elkészítünk egy formot, mely két listapanelt tartalmaz, és mindkét panelhez hozzáadunk egy feliratot. A feliratokat a panelek tetejéhez igazítjuk, a paneleket pedig a kliens területen elrendezzük. A két panel között egy Splitter (elválasztó) komponens helyezkedik el. Ezen eszközök segítségével a felhasználó könnyen méretezheti a formot. Ajánlom, hogy programjaink készítése során hasonló koncepciót kövessünk a későbbiekben is.

A program induláskor bemásolja az adatbázisok nevét az első listapanelbe:

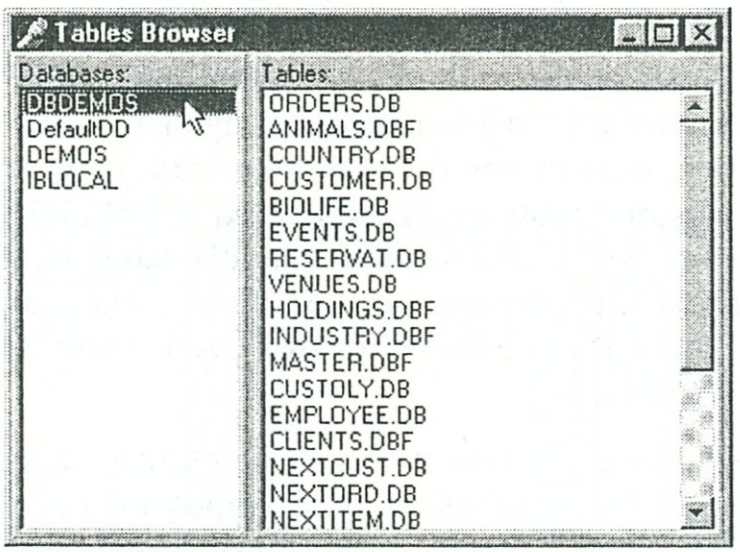
```
procedure TMainForm.FormCreate(Sender: TObject);
var
  DBNames: TStringList;
begin
```

```
    DBNames := TStringList.Create;
    Session.GetDatabaseNames (DBNames);
    ListBox1.Items := DBNames;
    DBNames.Free;
end;
```

Másik megoldásként közvetlenül is megadhatjuk a listapanel elemeit:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Session.GetDatabaseNames (ListBox1.Items);
end;
```

A GetDatabaseNames metódus kulcseleme a Session globális objektumnak. Delphiben készült adatbázis-kezelő alkalmazásokban mindig automatikusan létrejön egy TSession osztályú objektum, amelynek metódusait az uses DBTables utasítás kiadása után érhetjük el. A megnézett két példa közül az első feltölti a neki átadott stringlista objektumot, melyet az eljárás meghívása előtt létre kell hozni. Ezután egyszerre bemásolhatjuk a stringeket a listapanelbe. A másik megoldásban a panel Item tulajdonságát adjuk át paraméterként a GetDatabaseNames metódusnak.



16.19. ábra *Ha duplán kattintunk egy adatbázis nevén, a második listában megjelennek a tartalmazott táblák nevei*

Ha az első listapanel valamelyik elemére kattintunk, a másik panelben megjelennek a kiválasztott adatbázishoz tartozó táblák nevei. Az ezt végrehajtó programrészlet a TSession osztály egy másik, GetTableNames metódusát használja. Ennek öt paramétere van: Az adatbázis neve, egy, az adatok szűrésére használt string, két logikai érték, melyekkel beállíthatjuk, hogy megjelenjen-e a táblák fájlkiterjesztése, valamint a rendszertáblák, és az utolsó paraméter, amely egy

stringlista objektum, ebben kapjuk meg a táblák neveit. A következő sorokat hajtja végre a program, ha a felhasználó az első listapanel egyik elemén duplán kattint:

```
procedure TMainForm.ListBox1DblClick(Sender: TObject);
var
  CurrentDB: string;
begin
  CurrentDB := ListBox1.Items [ListBox1.ItemIndex];
  Session.GetTableNames (CurrentDB, '',
    True, False, ListBox2.Items);
end;
```

A program futásának eredményét a 16.19. ábrán láthatjuk. Ne feledjük tehát, hogy az ilyen típusú műveletek kulcsa a globális Session objektum, amely az éppen folyó adatbázis-műveletekről tartalmaz információkat.

Egy táblaböngésző program

Következő lépésként mutassuk meg a program által kilistázott táblák tartalmát. Ehhez szükségünk lesz egy második formra, amely egy DBGrid komponensen alapszik. A rács használatával könnyen megjeleníthetjük az adatokat, speciális vezérlők futási idő alatti készítése nélkül. A főform átláthatóságának megőrzése érdekében a rácsot a második formon helyezzük el. A program működésének lényege, hogy egy tábla kiválasztása esetén annak adatait egy új ablakban megjelenítjük. Minden olyan esetben, amikor egy új táblát jelenítünk meg, új formot kell létrehozzunk, ezért a program indulásakor ki kell kapcsolnunk az automatikus form létrehozást. Ezt a műveletet a projektfájl irányítja, változtatni ezeken a beállításokon a Project Options párbeszédablakban lehet. Válasszuk ki a **Forms** oldalt, és távolítsuk el a GridForm elemet az **Auto Create Forms** listából.

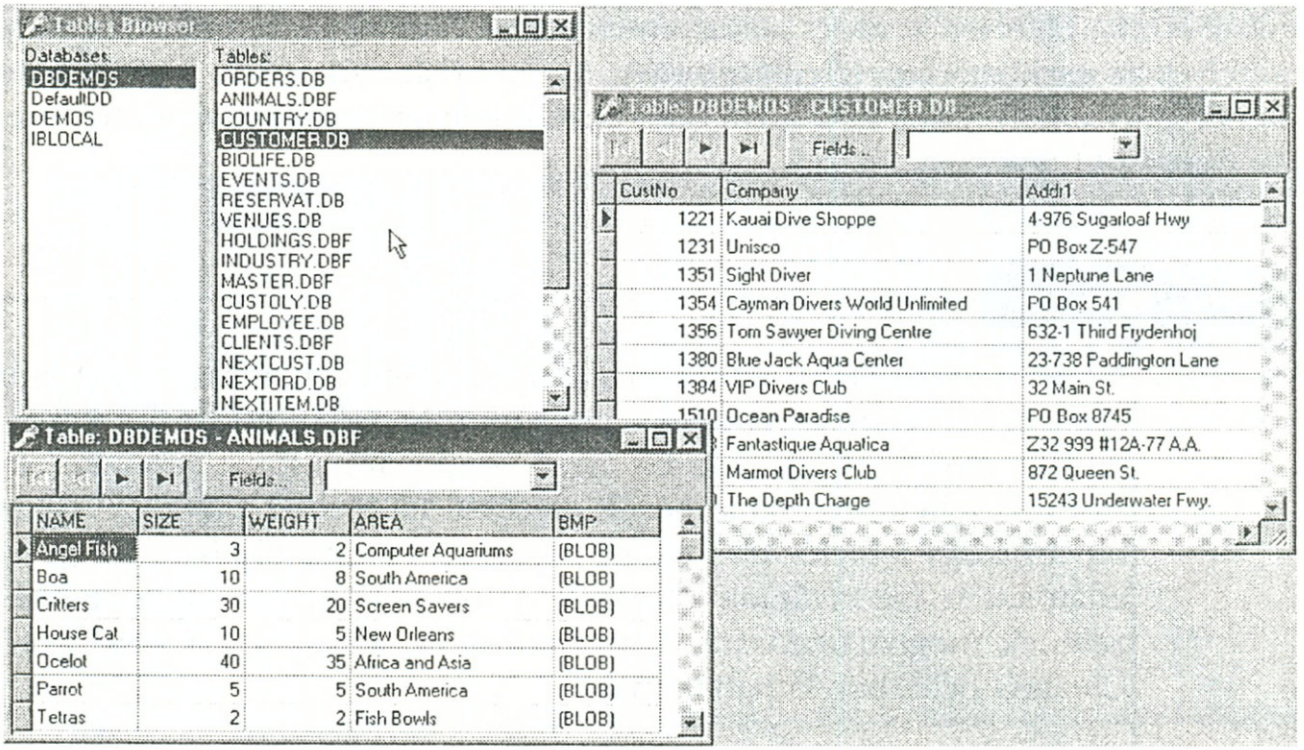
Annak, hogy nem egy globális objektummal dolgozunk, az az előnye, hogy több példányt is létrehozhatunk belőle futás közben. Ezek a formok nem modálisak lesznek, ami azt jelenti, hogy visszatérhetünk a főformhoz és megnyithatunk egy újat anélkül, hogy a már megnyitottat bezárnánk.

Minden alkalommal, amikor a felhasználó duplán kattint a második listapanel egy elemén, a program létrehoz egy TGridForm objektumot, összekapcsolja e form Table1 komponensét a megfelelő adatbázissal és táblával, majd megjeleníti a formot.

```
procedure TMainForm.ListBox2DblClick(Sender: TObject);
var
  CurrentDB, CurrentTable: string;
  GridForm: TGridForm;
begin
  CurrentDB := ListBox1.Items [ListBox1.ItemIndex];
```

```
    CurrentTable := ListBox2.Items [ListBox2.ItemIndex];
    GridForm := TGridForm.Create (self);

    {connect the table component to the selected
    table and activate it}
    GridForm.Table1.DatabaseName := CurrentDB;
    GridForm.Table1.TableName := CurrentTable;
    GridForm.Table1.Open;
    try
      {set the title and call a custom
      initialization method, then show the form}
      GridForm.Caption := Format ('Table: %s - %s',
        [CurrentDB, CurrentTable]);
      GridForm.FillFieldsCombo;
      GridForm.Show;
    except on Exception do
      GridForm.Close;
    end;
  end;
```

A program megjeleníti a tábla adatait tartalmazó új formot, ahogy azt a 16.20. ábrán is láthatjuk, ahol két ilyen form is szerepel. Felhívom a figyelmet arra, hogy a fenti program csupán létrehozza, de sohasem semmisíti meg a formokat. ezt a műveletet maga a form végzi el, úgy, hogy az OnClose esemény kezelőjének Action hivatkozási paraméterét caFree-re állítjuk.



16.20. ábra *A Tables program használatakor több rács alakú táblaböngészőt is megnyithatunk egyidőben*

A program a formok fejlécében megjeleníti a használt adatbázis és tábla nevét, valamint meghívja a form FillFieldsCombo eljárását. Ahogy nemsokára látjuk, ez egyszerűen bemásolja a mezők neveit egy kombinált listába. Lényegében véve ez a form inicializáló eljárása.

Ennek ellenére ezek a sorok nem kerülhetnek az OnCreate eseményhez, mivel a form létrejön a Table1 komponens értékeinek helyes beállítása előtt. Használhatnánk itt valami más eseményt (OnShow vagy OnActivate), de én inkább egy saját eljárást írtam, amelyet a főform hív meg. Nem kell, hogy az a tény, miszerint is a Windows eseményvezérelt, minket arra kényszerítsen, hogy minden alkalommal eseményvezérelt eljárásokat készítsünk. Előfordulhatnak olyan problémák, melyek megoldásához az egyszerű eljárások alkalmazhatók legkönnyebben, nem esemény váltja ki őket, egyszerűen meghívjuk a megfelelőt a kellő időben. Vessünk most egy pillantást a DBGrid form kódjára. Íme az a néhány sor, amely a FillFieldsCombo eljárás végrehajtásakor meghívódik.

```
procedure TGridForm.FillFieldsCombo;
var
  I: Integer;
begin
  for I := 0 to Table1.FieldCount - 1 do
    ComboBox1.Items.Add (Table1.Fields.FieldName);
end;
```

A kódrészletben láthatjuk, hogyan lehet elérni az adatbázisok harmadik fő elemét, az adatbázisok és a táblák listája után most már hozzáférhetünk a mezők listájához is. Ebben a konkrét példában eltároljuk őket egy kombinált listában, amelyet az eszköztárban jelenítünk meg. Továbbfejleszthetnénk a programot azzal, hogy elhelyezünk rajta adatfüggő vezérlőket, amelyek a kiválasztott mező típusától függenek.

Megjegyzés

Az adatkészlet Fields tömbjének használata mellett a FieldDefs tömb segítségével is megtudhatjuk a tábla mezőinek felépítését. Ez a tulajdonság nem adatokat, hanem mező meghatározásokat tartalmaz, és akkor is lehetővé teszi a mezők elérését, ha a tábla zárva van (igaz ugyan, hogy a FieldDefs Update metódusát meg kell hívnunk, ha az adatokat is be kívánjuk olvasni). A tulajdonságot a tábla szerkezetének meghatározására is használhatjuk, ahogy azt a következő CreateG példában láthatjuk majd.

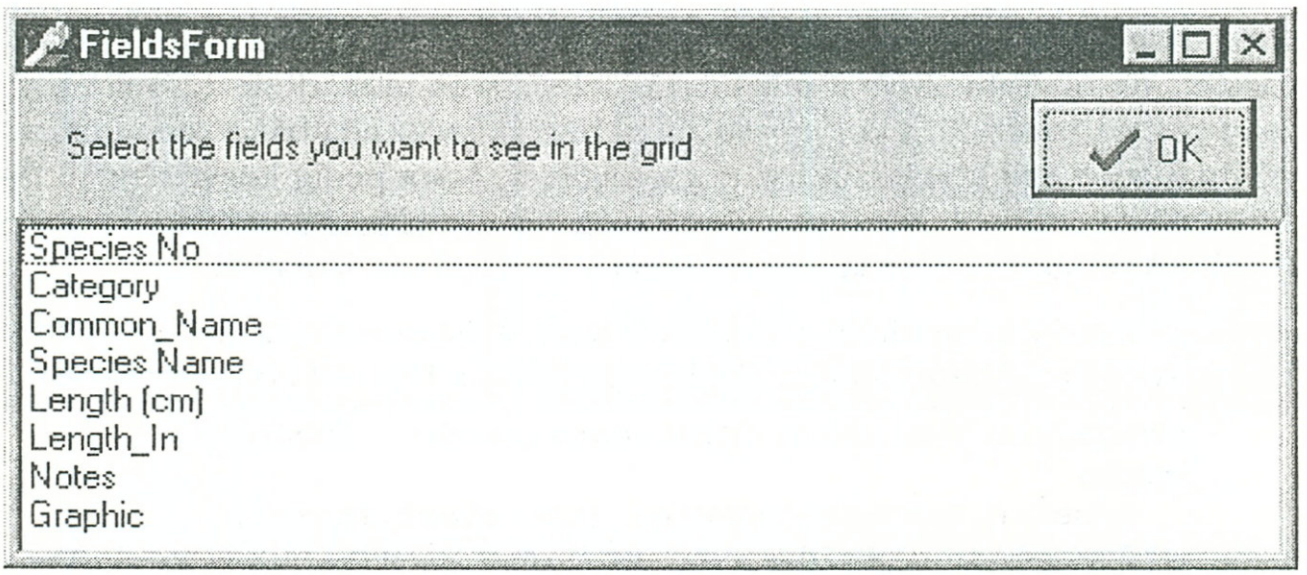
Mire is jó valójában ez a kombinált lista? E lista segítségével tudjuk meghatározni, hogy a formban mely mezők jelenjenek meg.

```
procedure TGridForm.ComboBox1Change(Sender: TObject);
begin
  {toggle the visibility of the field}
  Table1.FieldByName (ComboBox1.Text).Visible :=
    not Table1.FieldByName (ComboBox1.Text).Visible;
end;
```

A programrészletben a FieldByName metódus segítségével határozzuk meg a lista kiválasztott elemeit, és a Visible tulajdonsággal teszzük láthatóvá. Amikor egy mezőt kikapcsolunk, a táblához kapcsolt rácsból is azonnal eltűnik. Láthatjuk tehát, hogy a rácsot egyszerűen egy tulajdonság használatával megváltoztathatjuk. Ebben az irányban tovább is léphetünk, lehetőséget adhatunk a Columns tulajdonság értékeinek futásidejű megváltoztatására is.

Egy mezőszerkesztő program

Az eszköztárban elhelyezett kombinált lista működik ugyan, de egy nagy tábla (esetleg sok mező) használata esetén jelentősen lassítja munkánkat. E probléma kivédése érdekében készítsünk egy mezőszerkesztő formot. Ez lesz a Tables példa harmadik formja, neve FieldsForm. Ez a form modális, ezért használhatjuk globális objektumként minden alkalommal. Az új formhoz nem tartozik saját programrészlet. Mikor a rácsot tartalmazó form Fields gombjára kattintunk, a FieldsForm többválasztós listapaneljében megjelennek a Table1 komponens mezői. Ezzel párhuzamosan a program a listában bejelöli a látható mezőket, ahogy azt a 16.21. ábrán láthatjuk.



16.21. ábra *A listapanel segítségével kiválaszthatjuk a rács által mutatott mezőket*

A felhasználó e form segítségével választhatja ki, hogy mely mezőket kívánja szerkeszteni. A form bezárása után a program feldolgozza a megadott értékeket, és ezen információ alapján beállítja a megfelelő Visible tulajdonságokat. Íme az eljárás listája:

```
procedure TGridForm.SpeedButton1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to Table1.FieldCount - 1 do
  begin
    FieldsForm.FieldsList.Items.Add (
      Table1.Fields [I].FieldName);
    if Table1.Fields [I].Visible then
      FieldsForm.FieldsList.Selected [I]  := True;
  end;
  FieldsForm.ShowModal;
  for I := 0 to Table1.FieldCount - 1 do
    Table1.Fields.Visible [I] :=
      FieldsForm.FieldsList.Selected [I] ;
  FieldsForm.FieldsList.Clear;
end;
```

Ezzel a végére értünk ennek a bonyolult példának. Láthattuk, hogyan kell olyan adatbázis-kezelő alkalmazásokat készíteni, amelyek futási időben végeznek műveleteket, és azt is, hogy ez egy kissé bonyolult feladat. Többféleképpen is megkerülhettük volna ezeket a művelteket, például a program egyszerűsítésével.

A táblaböngésző felhasználói felületének továbbfejlesztése

Most, hogy kész vagyunk programunkkal, készítsünk el belőle egy másik változatot. Az új, Table2 című alkalmazás feladata és működése ugyanaz lesz, mint az előző verzióé, a különbség annyi lesz, hogy az adatbázisokat tartalmazó listapanelt egy kombinált listára cseréljük, a rácsot pedig közvetlenül a főformon helyezzük el. A form tartalmaz még egy elválasztót is a címke és a rács között. A program futásának eredményét a 16.22. ábrán láthatjuk.

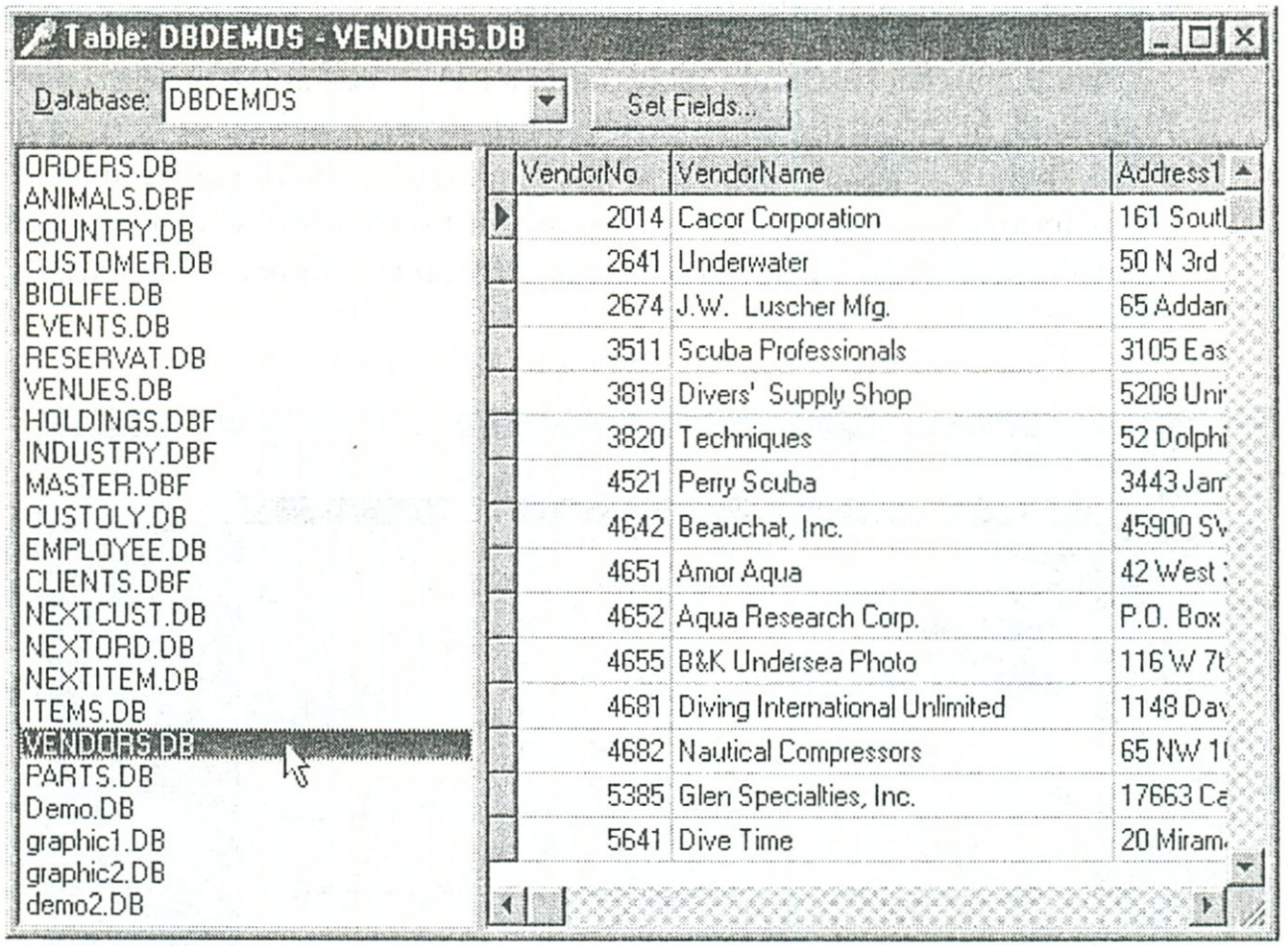
A program induláskor kitölti a kombinált listát, a listapanelt (az első elem kiválasztásával), majd megjeleníti a kiválasztott táblát a DBGrid komponensben.

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Session.GetDatabaseNames (ComboBox1.Items);
  ComboBox1.ItemIndex := 0;
  ComboBox1Change (self);
  ListBox1.ItemIndex := 0;
  ListBox1Click (self);
end;
```

Ezután a kezdeti értékadás végrehajtása után, és minden alkalommal, mikor a felhasználó új táblát választ, a rács tartalma változik, a helyes értékeknek megfelelően. A listapanel OnClick eseményének kódja a következő:

```
procedure TMainForm.ListBox1Click(Sender: TObject);
begin
  Table1.Close;
  Table1.DatabaseName := ComboBox1.Text;
  Table1.Tablename := Listbox1.Items
    [Listbox1.ItemIndex];
  Table1.Open;
  Caption := Format ('Table: %s - %s',
    [Table1.DatabaseName, Table1.Tablename]);
end;
```

A program eszköztárában megtalálhatjuk még a Fields feliratú gombot, amely megnyomáskor – a Tables példához hasonlóan – a Fields formot jeleníti meg.



16.22. ábra

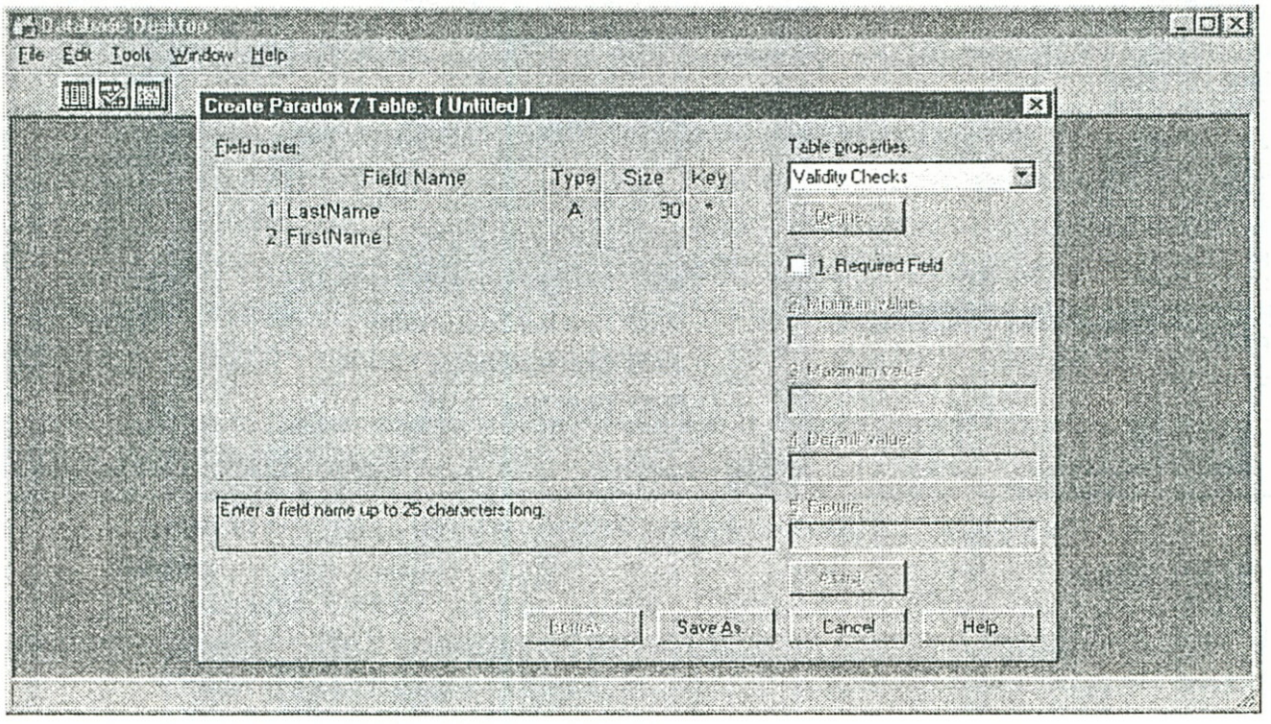
A Table2 program futtatási képe, a főformba ágyazott DBGrid komponenssel

Tábla létrehozása

Új alkalmazás készítésekor mindig neki kell ülnünk, és meg kell terveznünk a programunk által használt adatbázist, ami nem egy könnyű feladat. Még mérsékelten bonyolult feladatok esetében is az adatok kezelése hihetetlenül bonyolulttá tud válni. Ha még nem rendelkezünk elég tapasztalattal ezen a téren, akkor ajánlatos elolvasni egyet az adatbázis-tervezéssel foglalkozó könyvek közül, mielőtt összetettebb Delphi alkalmazások készítésébe fognánk.

Ha megterveztük adatbázisunkat, akkor neki foghatunk a használni kívánt táblák elkészítésének. Erre két lehetőségünk van: egyszerűen használhatjuk a Delphi Database Desktop alkalmazását, de természetesen programból is hozhatunk létre táblákat. Ha a Database Desktopot elindítottuk, a következő lépéseket kell végrehajtanunk:

- Válasszuk ki a **File | New | Table** menüparancsot.
- A megjelenő párbeszédablakban válasszunk egy táblatípust (például Paradox 7-et, vagy dBASE for Windows-t).
- Az ezután megjelenő Create Table (tábla létrehozása) ablak tartalma attól függ majd, hogy milyen táblatípust választottunk. A mezők neve és típusának meghatározása mellett lehetőségünk nyílik elsődleges és másodlagos indexek létrehozására, hivatkozások beállítására, érvényességek ellenőrzésére. A 16.23. ábrán láthatjuk a Paradox 7 típusú tábla létrehozásakor megjelenő ablakot.



16.23. ábra *A Paradox 7 típusú tábla létrehozásakor megjelenő Create Table párbeszédablak*

- Ha végeztünk, mentsük el az új táblát valamelyik mappánkba, vagy válasszunk számára egy BDE álnevet (aliast).
- Ezek után már megnyithatjuk a kész táblát, feltölthetjük értékekkel, vagy akár üresen is hagyhatjuk.

Táblák dinamikus létrehozása

Tegyük most fel, hogy egyszerű szerkezetű táblával dolgozunk, és lehetővé szeretnénk tenni a felhasználó számára több ilyen típusú tábla létrehozását. Egy Delphi alkalmazással ezt könnyedén megtehetjük. A következő program éppen erre a műveletre fog példát mutatni. Érdekessége az alkalmazásnak, hogy a könyvben első ízben egy grafikus mezőt is fogunk programunkban használni, és most először aknázzuk ki a Vágólap által nyújtott lehetőségeket (erről a témáról részletesen a 30. fejezetben olvashatunk majd).

Megjegyzés

A Delphi a grafikus, feljegyzés típusú, vagy BLOB mezőket ugyanúgy kezeli, mint a szokásosakat. Nekünk csupán a típushoz való szerkesztőprogramot kell kiválasztanunk, a dolog többi része rejtve, a színfalak mögött zajlik. Ha még nem láttuk a Delphihez adott grafikus példaprogramot, akkor próbáljuk meg most futtatni (a DEMOS\DB\FISHFACT mappában találhatók a szükséges fájlok), és vizsgáljuk meg, hogy működik.

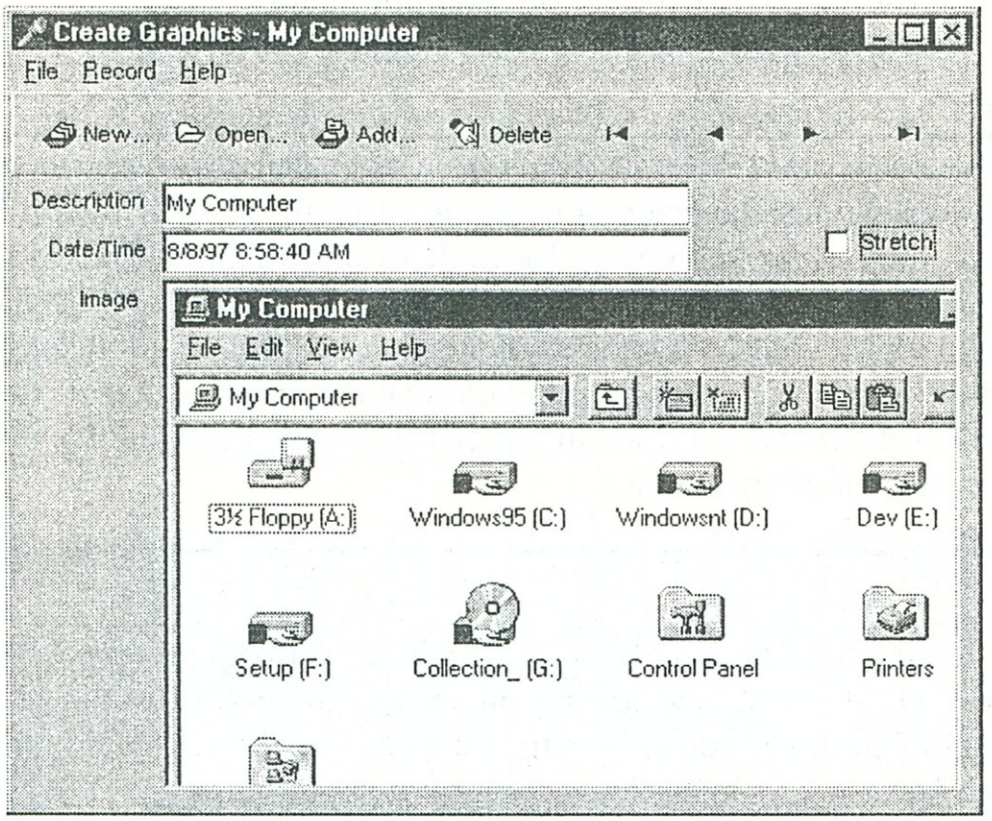
Az új példa címe CreateG lesz, ahol a G a grafikát jelenti. A program lehetővé teszi, hogy képekbe menthessük a képernyő pillanatnyi állapotát, és eltárolhassuk azokat egy táblában. Felhívom a figyelmet azonban, hogy ezek a táblák igen nagy méretűek lehetnek, így óvatosan bánjunk a programmal, ha kevés szabad lemezterülettel rendelkezünk.

Az alkalmazás minden futtatáskor rákérdez, hogy új táblát szeretnénk-e létrehozni, vagy egy meglévőt használni. Ha újat hozunk létre, akkor meg kell adnunk egy még nem létező táblanevet. Ha egy létezővel szeretnénk dolgozni, akkor pontosan meghatározott mezőkkel rendelkező táblát kell használnunk, olyat, ami tartalmazza a képeket, azok leírását, valamint elmentésük dátumát és idejét.

Ha elkészült a megfelelő tábla, a program megjeleníti főformját, ahogy azt a 16.24. ábra is mutatja. Most elmenthetjük a képernyő tartalmát úgy hogy azt a Vágólapra másoljuk. Ha megnyomjuk a PRINT SCREEN billentyűt, akkor az egész képernyő, az ALT+PRINT SCREEN kombináció használatával pedig az aktuális

ablak képe kerül a Vágólapra, bittérkép formátumban. Természetesen az alkalmazás más programokkal készített, és vágólapra kihelyezett képekkel is működik.

A kép Vágólapra helyezése után nyomjuk meg az **Add** (hozzáadás) gombot, vagy válasszuk ki a **Record | Add** menüpontot. Ekkor meg kell adnunk a kép leírását, a program beolvassa a rendszeridőt, kimásolja a Vágólap tartalmát, majd létrehozza az új rekordot. Ha ki szeretnénk törölni egy elemet a táblából, akkor használjuk a **Delete** (törlés) gombot, vagy válasszuk a **Record | Delete** pontot. Természetesen a form gombjaival kedvünkre lépkedhetünk a létrehozott rekordok között. Felhívom még a figyelmet a **Stretch** (nyújtás) jelölőnégyzetre, amely segítségével megadhatjuk, milyen méretben szeretnénk látni a képet.



16.24. ábra *A CreateG példa fő formja futás közben (a Stretch tulajdonságot éppen kikapcsoltam)*

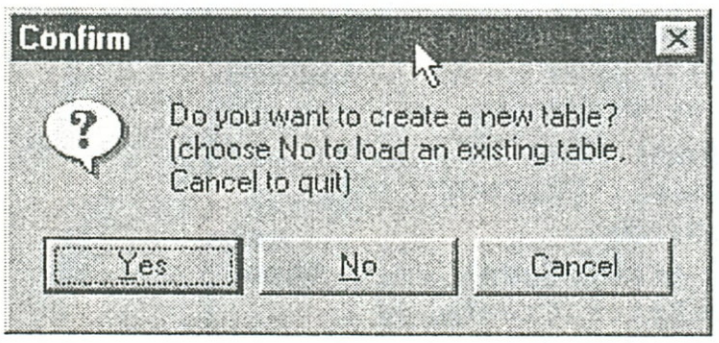
Ha kész a táblánk, készíthetünk egy újat is, a **New** gombbal, vagy a **File | New** paranccsal, esetleg megnyithatunk egy létezőt az **Open** gombbal, vagy a **File | Open** paranccsal. Természetesen tovább lehetne fejleszteni ezt a programot, például azzal, hogy a kép elkészítését automatikusan végezze, vagy hogy lehetőséget biztosítson a Vágólap tartalmának ellenőrzésére. Tovább azonban nem akarom bonyolítani a dolgot, a célnak nagyon jól megfelel ez a változat is.

A form és a kezdő programrészlet

Mindenekelőtt vessünk egy pillantást a fő formra, amely két Table komponenst tartalmaz! A második táblát (amely nem kapcsolódik adatforráshoz) egy kereső algoritmusban fogjuk használni. Megtalálható még néhány adatfüggő vezérlő a formon, egy eszköztár néhány gombbal és egy navigátorral, melyek Flat (lapos) tulajdonságát igazra állítottam. Ezen komponensek nem rendelkeznek semmilyen különleges tulajdonsággal, így leírásukkal itt nem foglalkozom.

A program induláskor megjelenít egy párbeszédablakot, ahogy azt a 16.25. ábrán láthatjuk. Ennek segítségével választhat a felhasználó az új tábla készítése, és egy már meglevő használata között. A FormCreate metódus által létrehozott ablak három választási lehetőséget kínál fel:

```
procedure TGraphForm.FormCreate(Sender: TObject);
var
  Code: Word;
begin
  Code := MessageDlg (
    'Do you want to create a new table?' +
    #13'(choose No to load an existing table,' +
    #13'Cancel to quit)',
    mtConfirmation, mbYesNoCancel, 0);
  if Code = idYes then
    New1Click (self)
  else if Code = idNo then
    Open1Click (self)
  else
    Application.Terminate;
end;
```



16.25. ábra *Az induláskor megjelenő dialógusablak*

A **Yes** gomb megnyomásakor a program a megfelelő eseménykezelő meghívásával létrehoz egy új táblát. A **No** választásakor egy létező tábla betöltésére nyílik lehetőségünk, a **Cancel**-el pedig kiléphetünk a programból. A program kódjának legfontosabb részeit a most következő New1Click és Open1Click eljárások tartalmazzák.

Új tábla létrehozása

Új táblát a Table komponens CreateTable metódusának meghívásával hozhatunk létre, de mielőtt ezt tennénk, be kell állítanunk a komponens néhány tulajdonságát. Meg kell adnunk a használt adatbázis nevét, a tábla nevét, a mezők neveit, és az index nevét.

Tervezési időben az adatbázis nevét a már ismert DBDEMOS álnévre állítottam. Ezt megváltoztathatjuk, de ebben az esetben a forráskód több helyén is ki kell cserélnünk ezt a nevet az újra. A tábla nevét a felhasználó adja meg, ehhez az InputQuery függvényt használjuk. Szükség van még arra is, hogy a beírt nevet leellenőrizzük, megtudjuk, hogy létezik-e már tábla a megadott néven; ez azért fontos, mert a CreateTable eljárás felülírja a már létező táblákat. Ezt az ellenőrzést a Session objektum egy metódusának meghívásával tudjuk elvégezni, hasonlóan az előző példához. Íme az ide vonatkozó kód első része:

```
procedure TGraphForm.New1Click(Sender: TObject);
var
  TableName: string;
  TbNames: TStringList;
begin
  {request the name of the new table from the user,
  raising an exception if Cancel is pressed}
  TableName := '';
  if InputQuery ('New Table',
    'Enter a new table name:', TableName) then
  begin
    if TableName = '' then
      raise EMyDatabaseError.Create (
        'Invalid table name');

    {if the table already exists in the DBDEMOS
    database, do not overwrite it}
    TbNames := TStringList.Create;
    Session.GetTableNames ('DBDEMOS', '',
      False, False, TbNames);
    if TbNames.IndexOf (TableName) >= 0 then
      raise EMyDatabaseError.Create (
        'Table already exists');
```

Az string lista IndexOf eljárása a string indexével tér vissza, ha az létezik, vagy –1-el, ha nem. Ha a megadott néven már van adatbázis, akkor a program egy kivételt vált ki, és a program futása megszakad. Az EMyExceptionClass egy új kivételosztály, amit magam hoztam létre:

```
type
  EMyDatabaseError = class (EDatabaseError)
  end;
```

Ugyanez a kivétel váltódik ki a New1Click metódusban, ha a felhasználó a Cancel gombra kattint az InputQuery párbeszédablakban. Ahogy azonban a következő néhány sorból láthatjuk, a kivétel csak akkor jön létre, ha a metódust a OnCreate eseménykezelő hívta meg. A program ezt a Sender paraméter segítségével határozhatja meg:

```
if InputQuery... then
  ...
else
  // if OnCreate called this method
  if Sender = self then
    raise EMyDatabaseError.Create (
    'Table creation aborted by the user');
```

Minden ilyen kivételt leellenőrzünk a FormCreate metódusban. Ha valami hiba történik a két lehetséges eljárás, a New1Click és az Open1Click végrehajtása során, akkor a program hibaüzenetet ad, és arra kéri a felhasználót, hogy válasszon újra a két lehetőség közül. Íme a FormCreate metódus teljes kódja:

```
procedure TGraphForm.FormCreate(Sender: TObject);
var
  Code: Word;
  Done: Boolean;
begin
  Done := False;
  while not Done do
  try
    Code := MessageDlg (
      'Do you want to create a new table?' +
      #13'(choose No to load an existing table,' +
      #13'Cancel to quit)',
      mtConfirmation, mbYesNoCancel, 0);
    if Code = idYes then
      New1Click (self)
    else if Code = idNo then
      Open1Click (self)
    else
      Application.Terminate;
    Done := True
  except
    on E: EMyDatabaseError do
      ShowMessage (E.Message);
  end; // end of try-except and while blocks
end;
```

Térjünk most vissza a New1Click metódus második részéhez. Ha a tábla még nem létezik, akkor a program létrehozhatja azt. Először tárolnunk kell a tábla nevét a Table1 komponensnél, valamint beállítanunk annak típusát ttParadoxra (vagy bármilyen már típusra). A következő lépés, amely egyben a

legbonyolultabb is, a tábla mezőinek létrehozása lesz. Szükségünk lesz a Table komponens FieldDefs tulajdonságára (ez tartalmazza a meződefiníciók listáját), és háromszor meg kell hívnunk az Add metódust, amellyel új mezőket adhatunk táblánkhoz. Az Add metódus IndexDefs tulajdonságát használjuk majd az elsődleges index meghatározására. Az ide tartozó programrészlet a következő:

```
{close the current table}
Table1.Close;

{set the name and type of the new table}
Table1.TableName := TableName;
Table1.TableType := ttParadox;

{define the three fields and the index}
with Table1.FieldDefs do
begin
  Clear;
  Add ('Description', ftString, 50, True);
  Add ('Time', ftDateTime, 0, False);
  Add ('Graphics', ftGraphic, 0, False);
end;
Table1.IndexDefs.Clear;
Table1.IndexDefs.Add('DescrIndex', 'Description',
  [ixPrimary, ixUnique]);

{create the table using the above definitions}
Table1.CreateTable;
Table1.Open;
Caption := 'Create Graphics - ' + TableName;
```

Az utolsó három sorból már tisztán látható, hogy a program a helyes értékek megadása után elkészíti a táblát a CreateTable metódussal, majd pedig megnyitja. Mivel az adatforrás és a többi adatfüggő vezérlők már össze vannak kapcsolva, a tábla megnyitása után megkezdhetjük a munkát.

Egy létező tábla kiválasztása a megfelelő mezők segítségével

A program második futtatásakor már kiválaszthatunk egy létező táblát is ahelyett, hogy újat hoznánk létre. A betöltő kódot az Open1Click metódus kezeli. Az első lépés a párbeszédablak listapaneljének táblanevekkel való feltöltése lesz.

Kezdetben azt gondoltam, hogy a lista elemeinek elérése a szokásos módon is történhet, de amikor ezzel megpróbálkoztam, folyton hibát jelzett a program. Vajon miért? A gond az, hogy a megírt programrészletet inicializáláskor meghívhatjuk a főform OnCreate metódusából. Biztosan emlékszünk még arra,

hogy általában egy projekt fájl többször is meghívja a CreateForm eljárást (pontosan annyiszor, ahány formot készítünk a program indulásakor. Láthatjuk hát, hogy műveleteink eredményeképpen az OnCreate metódus mindenképpen lefut.

Az előzők miatt történhet meg az, hogy a főform OnCreate eseménye közben a használt párbeszédablak még nem készült el, így a komponenseire való hivatkozás futásidejű hibához vezet. A fordító nem ismeri a programrészek végrehajtási sorrendjét, így nem is szűrheti ki a hibát. Problémánk megoldása az lesz, hogy kikapcsoljuk a TablesForm automatikus elkészítését a Project Options ablakban (a **Forms** oldalon), és saját magunk hozzuk azt létre a programban, még azelőtt, hogy bármely komponensét használnánk.

Itt az ideje, hogy továbblépjünk az eljárás lényegi része felé. A program (ugyanúgy, ahogy korábban) a Session objektum GetTableNames metódusával megszerzi az elérhető nevek listáját, de egy ellenőrzést hajt végre minden elemen, mielőtt elhelyezné azokat a listában. Az ellenőrzés úgy folyik, hogy a Table2 komponens felveszi a vizsgált tábla nevét, majd a program megvizsgálja azt, hogy helyesek-e a mezőnevek. Íme az eljárás első része:

```
procedure TGraphForm.Open1Click(Sender: TObject);

var
  TbNames: TStringList;
  I: Integer;
  TableFound: Boolean;

begin
  {create the form of the dialog box,
  before filling its list box with the table names}
  TablesForm := TTablesForm.Create (Application);

  {retrieve the list of tables from the database}
  TableFound := False;
  TbNames := TStringList.Create;
  Session.GetTableNames ('DBDEMOS', '',
    True, False, TbNames);

  {check if the table has the proper fields,
  that is, if it was created by this program.
  The code uses a secondary table object}
  for I := 0 to TbNames.Count - 1 do
  begin
    Table2.TableName := TbNames [I];
    Table2.FieldDefs.Update;
    if (Table2.FieldDefs.Count = 3) and
      (Table2.FieldDefs[0].DataType = ftString) and
      (Table2.FieldDefs[1].DataType = ftDateTime) and
```

```
        (Table2.FieldDefs[2].DataType = ftGraphic) then
      begin

        {table fields match: add the table to the list}
        TablesForm.Listbox1.Items.Add (Table2.TableName);
        TableFound := True;
      end;
    end;
```

16.26. ábra *A TablesForm párbeszédablak segítségével a felhasználó a megfelelő formátumú táblák közül választhat*

Ha a tábla megfelel a követelményeknek, bekerül a listapanel elemei közé, és az ezt jelző változót (TableFound) igazra állítjuk. Új tábla megtalálásakor a párbeszédablak megjelenik a képernyőn, ahogy azt a 16.26. ábrán is láthatjuk. Ha a párbeszédablakot bezárjuk az OK gombbal, akkor a listapanel éppen kiválasztott eleme bekerül a Table1 komponens TableName tulajdonságába, amelyet már korábban hozzákapcsoltunk a főform adatfüggő vezérlőihez. Ha egyetlen tábla sem felel meg a követelményeknek, akkor egy kivételt vált ki a program. Íme az Open1Click eljárás második része:

```
  {if no table was found, raise an exception}
    if not TableFound then
      raise EMyDatabaseError.Create (
        'No table with the proper structure');

    {otherwise, show the dialog box}
    TablesForm.ListBox1.ItemIndex := 0;
    if TablesForm.ShowModal = idOK then
    begin
```

```
      {if OK was pressed, open the table}
      Table1.Close;
      Table1.TableName := TablesForm.ListBox1.Items [
        TablesForm.ListBox1.ItemIndex];
      Table1.Open;
      Caption := 'Crete Graphics - ' +
        Table1.TableName;
    end
    else
      // if OnCreate called this method
      if Sender = self then
        raise EMyDatabaseError.Create (
          'Table selection aborted by the user');
end;
```

Rekordok hozzáadása és törlése

Ha a táblánk nyitva van, akkor már semmi akadálya annak, hogy új rekordokat hozzunk létre. Ezt minden olyan esetben el kell végeznünk, amikor a felhasználó az Add gombra kattint, vagy kiválasztja az ide vonatkozó menüparancsot. A gomb OnClick metódusa igen egyszerű lesz: megkérjük a felhasználót, hogy gépeljen be egy egyedi leírást a létrehozandó rekordról, majd beszúrjuk azt a tábla végére.

```
procedure TGraphForm.Add1Click(Sender: TObject);
var
  Descr: string;
begin
  if InputQuery ('New record',
    'Enter the description:', Descr) then
  begin
    Table1.Insert;
    EditDescription.Text := Descr;
    EditDate.Text := DateTimeToStr (Now);
    DBIMage.PasteFromClipboard;
    Table1.Post;
  end;
end;
```

Ne feledjük, hogy az adatok az adatfüggő vezérlőkbe, és nem közvetlenül a mezőkbe fognak kerülni. Esetünkben a PasteFromClipboard eljárást fogjuk használni arra, hogy a Vágólap tartalmát a DBImage komponensbe másoljuk. A főform rekordtörléssel, és a kép nyújtásával foglalkozó eljárásai nagyon egyszerűek:

```
procedure TGraphForm.Delete1Click(Sender: TObject);
begin
  if MessageDlg (
```

```
      'Are you sure you want to delete the current
record?',
        mtConfirmation, [mbYes, mbNo], 0) = idYes then
      Table1.Delete;
end;

procedure TGraphForm.CheckBox1Click(Sender: TObject);
begin
  DBImage.Stretch := CheckBox1.Checked;
end;
```

Hát ennyi az egész. Elkészítettünk egy fájlokat használó programot, amely rendelkezik a File | New és a File | Open menüparancsokkal. Igaz ugyan, hogy a program úgy viselkedik, mintha minden tábla egy külön fájl lenne, ez a valóságban nem mindig van így. A Paradox típusú adatbázisok ilyenek, vagyis minden tábla külön fájlban található, de ez nem jellemző a Microsoft Access, vagy az InterBase adatbázisokra, ahol egy fájl általában több táblát tartalmaz.

Többrekordú rács használata

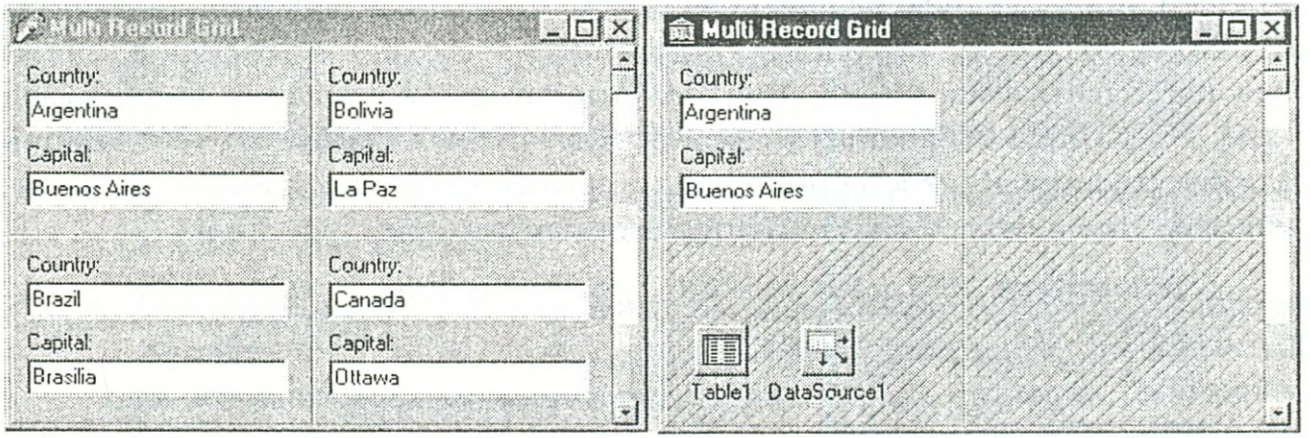
Eddig egy tábla adatainak megjelenítésére két módszert ismerhettünk meg: az egyik a rácsok, a másik a különböző adatfüggő vezérlők használata volt. Létezik azonban egy harmadik lehetőség is: a többrekordú rács (DBCtrlGrid) használata, amellyel egy form meghatározott területén elhelyezhetünk több adatfüggő komponenst, majd a rekordok számától függően automatikusan megtöbbszörözhetjük ezeket a vezérlőket.

Vegyük most sorra, mi is szükséges új, Multi1 című példának elkészítéséhez. Egy üres formból kell kiindulnunk, el kell rajta helyeznünk egy Table és egy DataSource komponenst, majd hozzá kell kapcsolnunk ezeket a COUNTRY.DB táblához. Adjunk hozzá még egy DbCtrlGrid komponenst a programhoz, állítsuk be a méretét, a sorainak és oszlopainak számát, majd helyezzünk el két szerkesztőmezőt az összetevőn, és kapcsoljuk ezekhez a tábla Name és Capital mezőit. A DBEdit komponenseket a Fields Editor segítségével is hozzáadhatjuk formunkhoz. Tervezési időben a rács aktív részén tudunk csak dolgozni (16.27. ábra, jobb oldal), futtatáskor viszont megjelenik a megfelelő számú másolat (16.27. ábra, bal oldal).

A DBCtrlGrid komponens és társainak legfontosabb tulajdonságai a következők:

```
object Form1: TForm1
  Caption = 'Multi Record Grid'
  object DBCtrlGrid1: TDBCtrlGrid
    ColCount = 2
    DataSource = DataSource1
    RowCount = 2
    object DBEdit1: TDBEdit
```

```
        DataField = 'Name'
        DataSource = DataSource1
      end
      object DBEdit2: TDBEdit...
    end
    object Table1: TTable
      Active = True
      DatabaseName = 'DBDEMOS'
      TableName = 'COUNTRY.DB'
    end
    object DataSource1: TDataSource
      DataSet = Table1
    end
  end
```



16.27. ábra *A Multi1 példa DBCtrlGrid komponense tervezési (jobbra) és futási (balra) időben*

Láthatjuk, milyen egyszerű a sorok és oszlopok számának állítása. Minden alkalommal, mikor átméretezzük a vezérlőt, a tartalmazott panelek szélessége és magassága ezen értékeknek megfelelően fog változni. Sajnos arra nincs lehetőségünk, hogy a komponens az ablak átméretezésekor automatikusan lefedje annak területét.

Rácspanelek mozgatása

Az előző példát folytatva próbáljuk meg most a FormResize metódussal átméretezni rácsunkat. Írjuk be egyszerűen a következő sorokat (a példa címe Multi2 lesz):

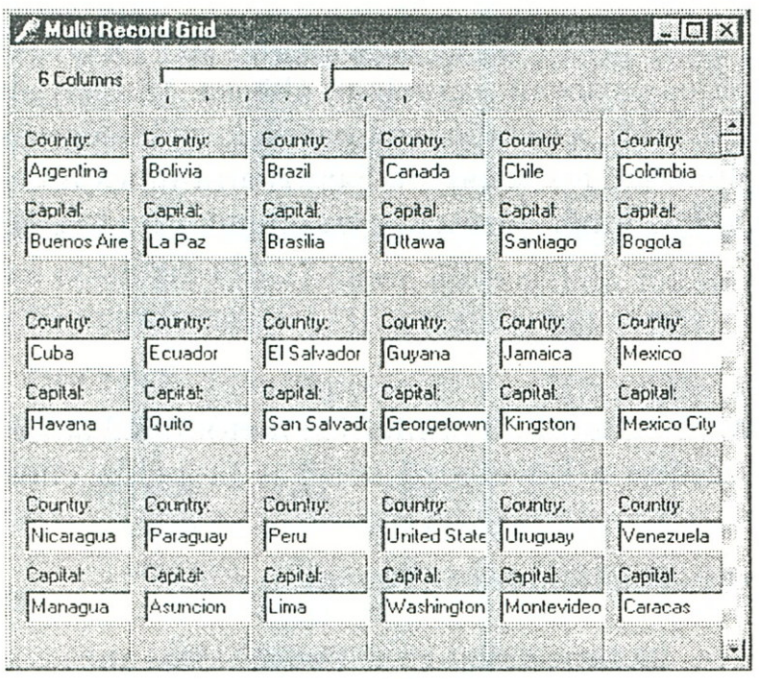
```
procedure TForm1.FormResize(Sender: TObject);
begin
  DBCtrlGrid1.Height := ClientHeight - Panel1.Height;
  DBCtrlGrid1.Width := ClientWidth;
end;
```

Ez ugyan működik, de nem úgy, ahogy én szeretném. Az lenne az igazi, ha nem a panelek mérete, hanem azok száma változna, ahogy az ablakot méretezzük. Először is számoljuk ki, mennyi a panelek minimális magassága, és hogy hány ilyen elemet tudunk átméretezés után megjeleníteni. A Multi2 példa FormResize metódusát kiegészítettem egy új utasítással, így ez most a következőképpen néz ki:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  DBCtrlGrid1.RowCount :=
    (ClientHeight - Panel1.Height) div 100;
  DBCtrlGrid1.Height := ClientHeight - Panel1.Height;
  DBCtrlGrid1.Width := ClientWidth;
end;
```

Ahelyett, hogy ugyanezt elvégeznénk a rács oszlopaira is, úgy döntöttem, hogy inkább egy TrackBar (vezérlősáv) komponenst használok az oszlopok számának meghatározására (16.28. ábra). A program a sáv értékének megfelelően (ami 2 és 10 között változhat) állítja a rács oszlopainak számát. Ha visszaállítjuk az oszlopszámot a kezdeti értékre, akkor természetesen azok mérete is visszaváltozik. Íme a vezérlősáv OnChange eseménykezelője:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  Label1.Caption := Format (
    '%d Columns', [TrackBar1.Position]);
  DBCtrlGrid1.ColCount := TrackBar1.Position;
  DBCtrlGrid1.Width := ClientWidth;
end;
```



16.28. ábra *A Multi2 példa futás közben, megnövelt oszlopszámmal*

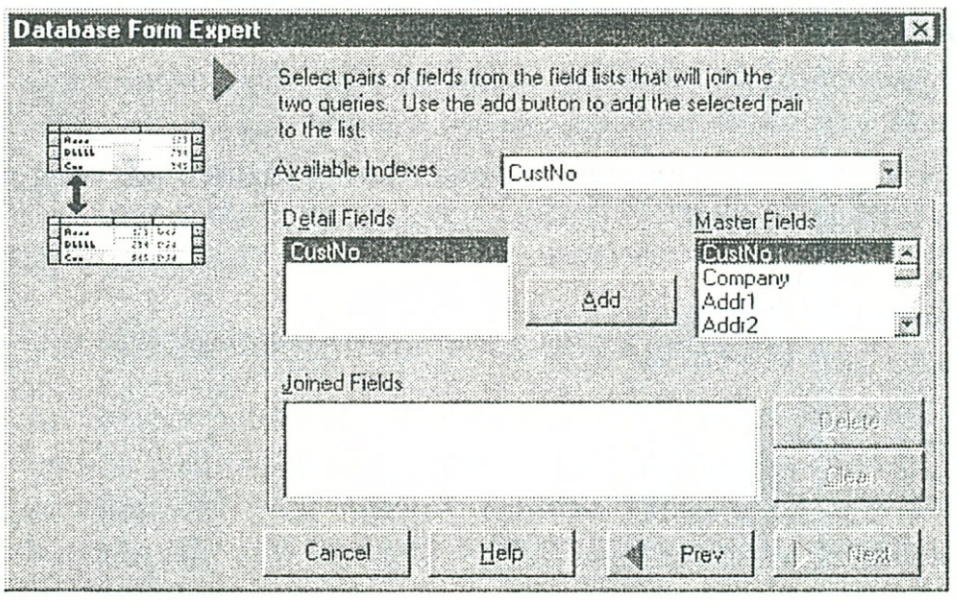
Kapcsolttáblás form készítése varázslóval

A Database Form Varázsló induláskor megkérdezi, hogy egy egyszerű táblán (single table), vagy kapcsolt táblán (master detail) alapuló formot akarunk készíteni. A második pont arra nyújt lehetőséget, hogy két táblát használjunk fel programunkban, amelyek között egy-sok kapcsolat áll fönn. Ezt a fajta kapcsolatot a megfelelő SQL utasítással, vagy a tábla tulajdonságainak beállításával érhetjük el, de a legegyszerűbb, ha hagyjuk, hogy a Varázsló mindent elvégezzen helyettünk. Pontosan erre fogunk példát látni a következőkben.

A Delphi beépített példaadatbázisait használjuk, így sajnos nincs túl sok választási lehetőségünk arra vonatkozóan, hogy mely táblákkal dolgozzunk. Példánkban a Customer (vevő) és az Orders (megrendelések) táblákat fogjuk felhasználni (ezeket egyébként más Delphi példaprogramok is előszeretettel alkalmazzák).

A program elkészítéséhez a következő lépéseket kell végigjárnunk:

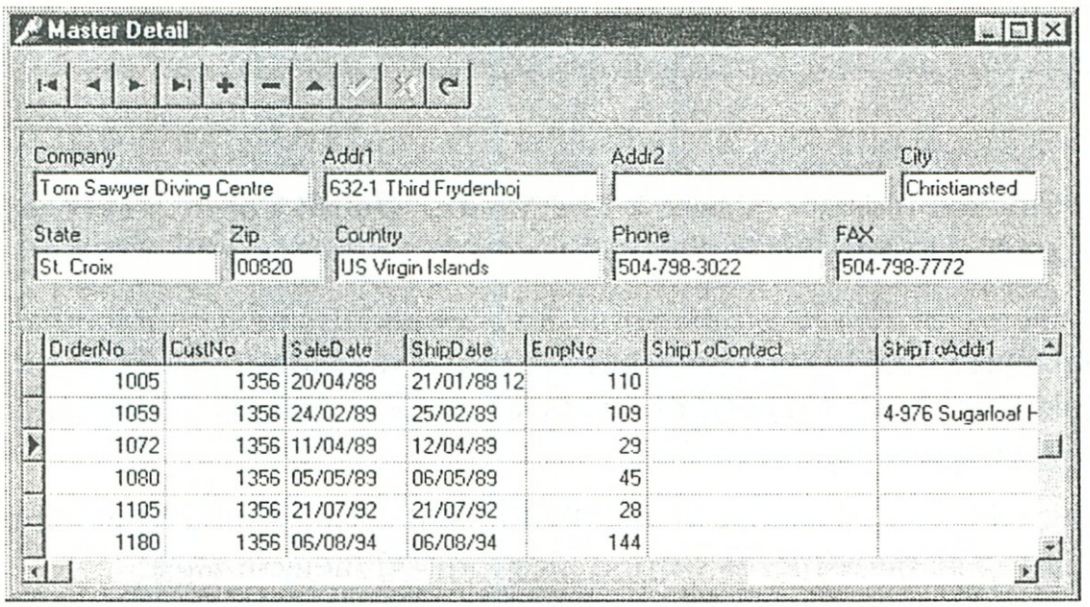
- Kezdjünk egy új projektet, és indítsuk el a Database Form Varázslót.
- A Varázsló első oldalán válasszuk ki a **Master Detail Form** pontot, és a TTable objektumot.
- Válasszuk a DBDEMOS gyűjtemény CUSTOMER.DB tábláját főtáblának (master table), majd válasszunk ki néhányat a megadott mezők közül, és adjuk meg, hogy függőleges (vertical), vagy vízszintes (horizontal) elrendezést szeretnénk használni.
- Kapcsolt táblának válasszuk ki az ORDERS.DB-t, válasszunk ki néhány mezőt, és az alapértelmezés szerinti rács formájú megjelenítést.



16.29. ábra *A Database Form Varázslóban ezen az oldalon határozhatjuk meg a használt táblák közötti kapcsolatot*

- Most érkeztünk a Varázsló legfontosabb oldalára (lásd 16.29. ábra). Válasszuk ki az indexek közül a CustNo-t, és jelöljük ki ugyanezt a mezők két listájában.
- Az **Add** gombbal hozzuk létre a kapcsolatot.
- Most már elkészíthetjük a formot (anélkül, hogy bejelölnénk a **Form and DataModule** opciót), és akár azonnal futtathatjuk is.

A 16.30. ábrán láthatjuk, hogy hogyan is néz ki a program futás közben. A képen látható formnak ugyan megnöveltem egy kicsit a méretét, és a komponenseket is átrendezgettem, de a program működését ez nem befolyásolja. Minden alkalommal, amikor egy új vásárlót választunk ki, a rácsban megjelennek a vevőhöz tartozó megrendelések.



16.30. ábra *A Database Form Varázsló által készített MastDet példa*

Hogyan is működik ez a program? Nos, igen egyszerűen: nézzük csak meg a két Table komponens tulajdonságait!

```
object Table1: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'customer.db'
end
object Table2: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'orders.db'
  IndexFieldNames = 'CustNo'
  MasterFields = 'CustNo'
  MasterSource = DataSource1
end
```

A második tábla rendelkezik egy adatforrással (azzal, amivel az első tábla össze van kapcsolva), és hivatkozik egy adott mezőre, ami alapján az összekapcsolást végzi.

A kapcsolttáblás szerkezet használata lekérdezésekben

Az előző példában a program elkészítésénél két táblát használtunk fel, ám arra is lehetőségünk van, hogy az összekapcsolást SQL parancsokkal végezzük el. Ez szintén nem lesz nehéz feladat, hiszen a Database Form Varázsló elkészíti az ehhez szükséges programot számunkra. Futtassuk újra a varázslót, de most a TForm helyett a TQuery-t válasszuk.

Ebben a példában az ORDERS.DB táblát az ITEMS.DB-vel kapcsoltam össze. A második tábla egy rendelés tárgyainak részletes leírását tartalmazza. A két táblát az OrderNo mezővel lehet összekapcsolni. Ha futtatjuk a programot (a címe Orders), azt fogjuk látni, hogy ugyanúgy működik, mint az előző. A példa érdekessége most a második lekérdezés objektum SQL parancsa:

```
select
  items."OrderNo",
  items."ItemNo",
  items."PartNo",
  items."Qty"
from
  items
where
  "items"."OrderNo" =: "OrderNo"
```

Láthatjuk, hogy ez az SQL parancs egy paramétert is használ, az OrderNo-t. Ez a paraméter közvetlenül az első lekérdezéshez kapcsolódik, hiszen a Query2 DataSource tulajdonsága DataSource1-re van állítva, amely pedig a Query1-hez kapcsolódik. Más szóval, a második lekérdezés az első lekérdezés vezérlőjeként szolgál. Minden olyan alkalommal, amikor az első adatforrás aktuális rekordja megváltozik, a Query2 komponens is változik ugyanúgy, mint minden más DataSource1-hez kapcsolt komponens. Ebben az esetben a kapcsolathoz használt mező neve megegyezik a lekérdezés paraméterének nevétől.

Választás kombinált lista segítségével

Az Orders példát természetesen tovább lehet fejleszteni (az eredeti változatot nem is találhatjuk meg a CD-n). Az új verzióban két oszlopba rendeztem a vezérlőket, és még néhány apróbb változtatást végeztem a program megjelenésén, de ami ezeknél sokkal lényegesebb: megoldottam a form egy nagy problémáját.

A program eredeti verziójában számokkal kellett a vevőkre hivatkozni, ha meg akartunk tekinteni egy rekordot, vagy átszerkeszteni annak tartalmát. Ez

azonban nehezen kezelhető, és nem túl felhasználóbarát dolog, hiszen így külön nyilván kellene tartanunk, hogy melyik szám melyik vevőt jelenti. Ezt azonban megteheti helyettünk a számítógép is: a helyezzünk a formra egy DBLookupComboBox-ot! Ezt a komponenst két adatforráshoz is hozzákapcsolhatjuk, az egyik a konkrét, a másik pedig a megjelenítendő adatokat fogja tartalmazni. A példában a DataSource1 CustNo értékét akarjuk használni, de a CUSTOMER.DB táblából származó adatokat szeretnénk mutatni.

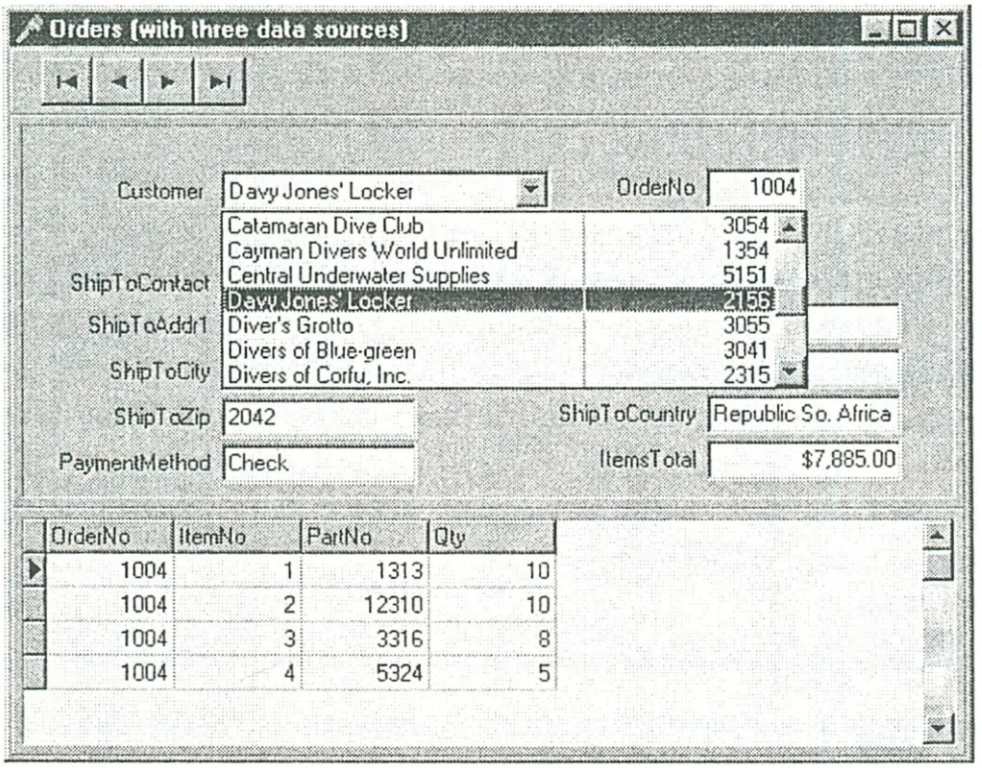
Távolítsuk el a vevőszámhoz kapcsolt szerkesztőmezőt a formról, és helyettesítsük egy DBLookupComboBox és egy DBText komponenssel! A DBText egy felirat, amit nem lehet futás alatt szerkeszteni. Ezután adjunk a programhoz egy új adatforrást (DataSource3), és kapcsoljuk azt a CUSTOMER.DB táblához! A kombinált lista néhány tulajdonságát be kell még állítanunk:

```
object DBLookupComboBox1: TDBLookupComboBox
  DataField = 'CustNo'
  DataSource = DataSource1
  KeyField = 'CustNo'
  ListField = 'Company'
  ListSource = DataSource3
end
```

Az első két tulajdonság szokás szerint a fő kapcsolatot határozza meg. A másik három a másodlagos forrásra (ListSource), a kapcsoláshoz használt mezőre (KeyField), és a megjelenített információkra (ListField) vonatkozik. Egy egyszerű mezőnév helyett akár többet is megadhatunk: a listában csak az első mező jelenik majd meg, de ha a DropDownWidth (legördülési szélesség) értékét elég nagyra állítjuk, akkor a megjelenő lista több oszlopnyi információt fog tartalmazni. A programot a 16.31. ábrán láthatjuk futás közben.

Tipp

Ha a DBLookupComboBox komponenshez kapcsolt tábla indexét a Company mezőre állítjuk, akkor a listában a vállalatok nevei ábécé sorrendben, és nem a vevőszámok szerint fognak megjelenni. A programban magam is hasonlóan jártam el.



16.31. ábra *Az Orders példa futás közben, a több mezőt is tartalmazó DBLookupComboBox komponenssel*

Mi a helyzet a program kódjával? Nos, az nincsen neki. Minden tökéletesen működik, és csak a megfelelő tulajdonságokat kellett beállítanunk, a három összekapcsolt adatforrásnak nincs szüksége saját kódra.

Keresés a rácsban

Most, hogy elhelyeztünk egy keresőmezőt (a DBLookupComboBox használatával) a formon, lehetőségünk nyílik arra, hogy egy legördülő listát hozzunk létre a rácson (ezt az oszlopszerkesztő PickList gombjával hajthatjuk végre). A kereső lista hozzáadásával azonban, amely egy másodlagos adatforráshoz fog csatlakozni, az előzőektől eltérő módszert kell alkalmaznunk: a Fields Editor-ral kell egy keresőmezőt készítenünk.

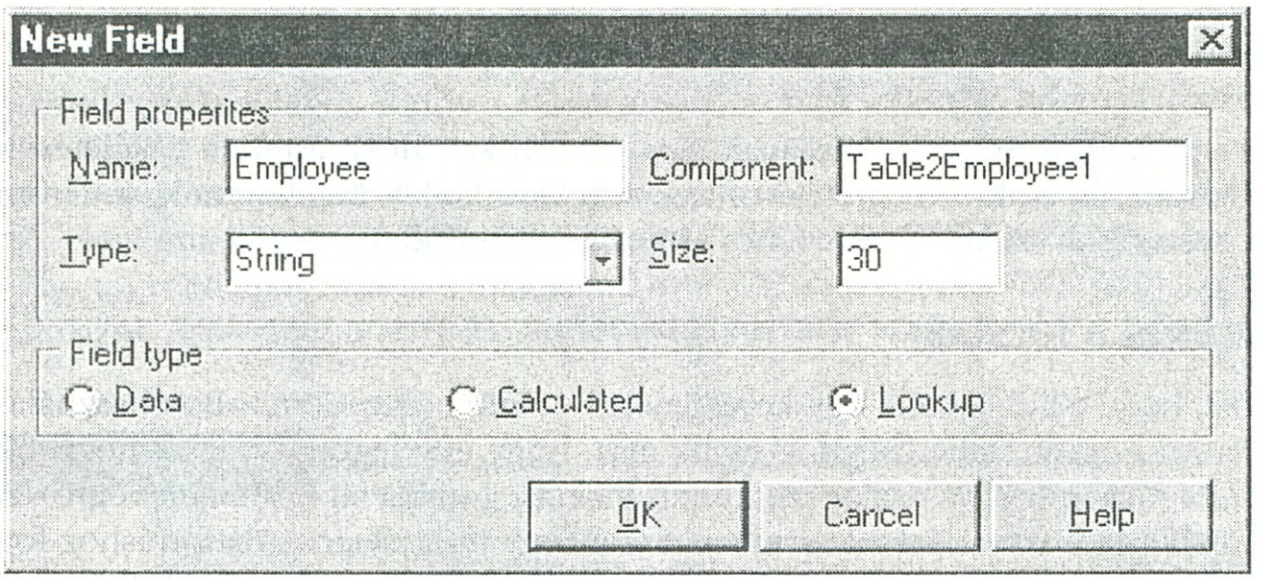
A példánkat a MastDet formból készítjük el, és MastDet2-nek fogjuk nevezni. A program eredeti verziójában annak a dolgozónak a számát mutatta a rács, aki felvette az adott megrendelést. Jelenítsük meg itt most inkább az alkalmazott nevét, és tegyük lehetővé, hogy ezt az értéket egy listából lehessen kiválasztani!

A cél elérése érdekében hozzáadtam a formhoz egy táblát és egy adatforrást, melyek az EMPLOYEE.DB táblához kapcsolódnak. Megnyitottam a Fields Editor-t a második (vagyis a rendeléseket tartalmazó) táblával, és hozzáadtam

az összes mezőt. Kiválasztottam az EmpNo mezőt, és a Visible (látható) tulajdonságát hamisra állítottam, így ez nem fog látszódni a rácsban (nem távolíthatjuk el teljesen, hiszen az Employee táblára való kereszthivatkozáskor szükségünk lesz rá).

Most már elkészíthetjük a keresőmezőt. Ha követtük az előbbi utasításokat, akkor a Fields Editor helyi menüjének **New Field** parancsával létrehozhatunk egy új mezőt a megjelenő dialógusablak segítségével, ahogy azt a 16.32. ábra mutatja. Egyszerűen írjunk be egy nevet (például azt, hogy Employee), és jelöljük be a Lookup opciót. Már csak egy dolgunk van hátra: be kell állítanunk a mező tulajdonságait:

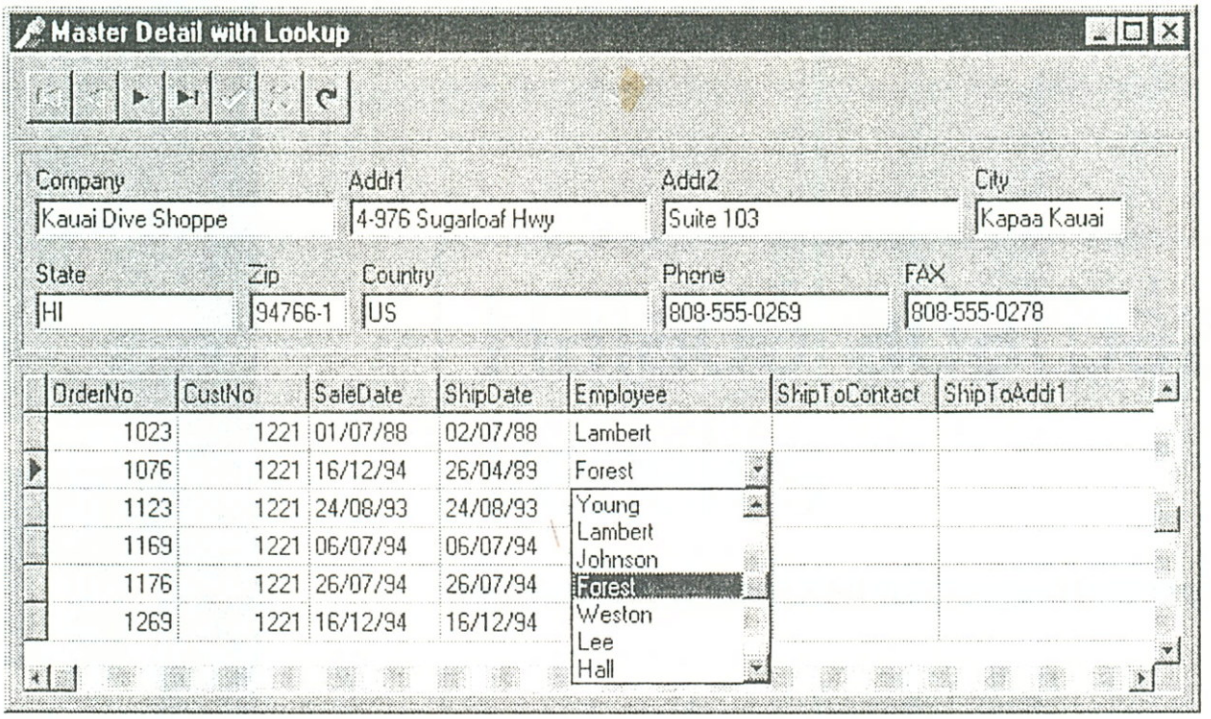
```
object Table2Employee: TStringField
  FieldName = 'Employee'
  Lookup = True
  LookupDataSet = Table3
  KeyFields = 'EmpNo'
  LookupKeyFields = 'EmpNo'
  LookupResultField = 'LastName'
end
```



16.32. ábra

A keresőmező létrehozására használt dialógusablak

Ez minden, most már helyesen működik programunk (16.33. ábra). Felhívom a figyelmet még arra, hogy nincs szükségünk arra, hogy a rács Columns tulajdonságán állítsunk, hiszen az alapértelmezésben tartalmazza a hét oszlopot, és a listát. Természetesen azért megváltoztathatjuk ennek értékét, ha nem vagyunk a program megjelenésével megelégedve.



16.33. ábra *A MastDet2 példa futás közben. A legördülő mező értékei egy másik táblából származnak*

Mi következik?

Ebben a fejezetben rengeteg példát láttunk arra, hogy miképpen működik az adatbázis-kezelés a Delphiben. Megtanultuk, hogyan lehet mozogni az adatok között, hogyan változtathatunk az értékeken, hogyan törölhetünk a rekordok közül, és hogyan készíthetünk újakat. Az utolsó részben megismerhettünk néhányat a magasabb szintű tulajdonságok közül, táblákat készítettünk, és különböző felhasználói felületeket építettünk fel. Még arra is láttunk egy példát, hogy hogyan lehet grafikus mezőket használni adatbázis-kezelő alkalmazásoknál, legvégül pedig táblák összekapcsolásával foglalkoztunk.

Természetesen még messze nem merítettük ki a Delphi adatbázisokkal kapcsolatos lehetőségeit. Ennek a fejezetnek csupán az volt a célja, hogy megismerkedjünk az ilyen típusú programok elkészítésének alapjaival. Nem esett szó adatbázisok tervezéséről, és nem is használtunk túl bonyolult táblákat, mert azt hiszem, ez a téma már túllépi könyvünk kereteit. A következő fejezetben (ami szintén az adatbázisokkal foglalkozik) egy általánosabb képet szeretnék nyújtani a témáról: szó esik a kliens/szerver stílusú programozásról, a Visual Query Builder-ről (Vizuális Lekérdezés-szerkesztő), az adatmodulokról, a Data Dictionary-ról (adatkönyvtár), és sok minden másról is.

17. FEJEZET

Haladó adatbázis-kezelés

E fejezet tartalma:

- Kapcsolat létesítése SQL szerverrel
- Meglévő program átírása SQL szerverre
- Tábla-összekapcsolások a Visual Query Builderrel
- Adatmodulok
- Adatmodul több nézethez
- Az adatszótár
- Tranzakciókezelés
- Adatbázishibák kezelése

Haladó adatbázis-kezelés

A 16. fejezetben több példát is láthattunk az adatbáziskezelés megvalósítására. Minden ilyen példaprogram dBase vagy Paradox – tehát viszonylag megszokott – táblákból vette az adatokat. Ez természetesen megfelelő egyszerűbb programoknál, de ha megbízható illetve biztonságos alkalmazást akarunk írni, akkor megfontolandó egy SQL szerver használata (az SQL rövid áttekintését a B függelékben találhatjuk meg). Az SQL szerver lehet bármelyik hálózatba kötött szervergépen, vagy akár ugyanazon a helyi gépen, amelyiken a programunkat fejlesztjük. A Delphi rendszerhez a Borland SQL szerverének Windows-os változatát, az InterBase-t szállítják.

Ebben a fejezetben az SQL szerverek használatának néhány részletét fogjuk bemutatni. Ha nem a Delphi Client/Server Suite változatával rendelkezünk, akkor kísértést érezhetünk arra, hogy ezt a részt kihagyjuk. De néhány eszköz amit itt is bemutatunk (például a helyi InterBase szerver) minden Delphi változatnak része. A fejezet második fele olyan további haladó adatbáziskezelő témákkal foglalkozik, mint az adatmodulok vagy az adatszótár. Ezek a vonások nem tartoznak szigorúan csak a kliens/szerver fejlesztéshez, bár nagy hasznunkra válhatnak ezen a területen is.

Kapcsolat létesítése SQL szerverrel

A helyi SQL szerver használható mind fejlesztésre, mind célplatformként. Ha célplatformként akarjuk használni, akkor fel kell installálnunk az InterBase szervert (persze csak ha van hozzá licensz jogunk) a saját programunkkal és a hozzá szükséges BDE könyvtárakkal együtt.

Megjegyzés

Vegyük figyelembe, hogy a helyi InterBase szerver csak egyfelhasználós rendszer, nem teljes, sokfelhasználós SQL kiszolgáló. A helyi InterBase szerver hasznos lehet olyan alkalmazások telepítésénél, melyek különálló (nem hálózatba kötött) számítógépen futnak, anélkül, hogy lemondanánk az SQL lekérdezések előnyeiről vagy az alkalmazás egy másik, helyi táblákat használó változatát is meg kellene írnunk. A Borland az InterBase teljes, Windows NT vagy UNIX alatt futó változatát is árulja.

A helyi InterBase szervert gyakran csak fejlesztési célokra használjuk. Ahelyett, hogy rögtön a hálózaton fejlesztenénk, ezt a helyi gépen is megtehetjük, majd a végén egyszerűen átállítjuk a céladatbázist. Ez lehet a közbülső lépés a lokális, fájlokat használó verzió (a programunk egy változata) és a teljesen elkészült kliens/szerver (a kész alkalmazás) között.

Egy távoli szerverre kapcsolódó Delphi alkalmazás fejlesztése alig különbözik egy helyi táblákat használó Delphi alkalmazástól. Ha már egyszer készen van a szerveren az adatbázis és a megfelelő táblákat már megadtuk, akkor a távoli szervert szinte zökkenőmentesen tudjuk használni a fejlesztői környezetből. Persze azért vannak különbségek. Például a felhasználónak általában be kell jelentkeznie az adatbázis eléréséhez. Az SQL szerver jelenlétének másik kulcsfontosságú része a hatékony tranzakciókezelés lehetősége (ugyanis a 32 bites Borland Database Engine a Paradox és dBase táblákra csak viszonylag egyszerű tranzakciókezelést valósít meg). További lehetőség a tárolt eljárások végrehajtása és a hatékonyság növelése táblák helyett lekérdezések használatával. Adott SQL szerver esetén indexelés és különböző további optimalizációs technikák is jelen lehetnek.

Ezeknek a különbségeknek egy része áthidalható a Delphi adatbáziskomponenseinek használatával. Például a Database és a Session komponensek lehetővé teszik az adatbázisba való belépés műveletének testreszabását valamint annak meghatározását, hogyan viselkedjen egy tranzakció, ha egy másik tranzakció szimultán módon ugyanazokat a táblákat éri el (*tranzakció-elválasztás* szintje). Bár ezen komponenseket nem kötelező programunkba építeni, használatuk megkönnyítheti a szerverkörnyezet irányítását.

Az SQL szerver egy másik jellemzője a StoredProc komponens. Ez lehetővé teszi egy, a szerver adatbázisában tárolt eljárás végrehajtását, esetlegesen paraméter átadásával illetve visszatérési érték fogadásával. Az adatbázisok kezelésére egy újabb hasznos komponens a BatchMove, amivel programunk rekordok egész csoportjait illetve komplett táblákat másolhat, hozhat létre vagy töröllhet akár két különböző adatbázisban is. Ennek egy érdekessége, hogy ezek közül az egyik tábla lehet helyi, a másik pedig az SQL szerveren.

Első InterBase alkalmazásunk

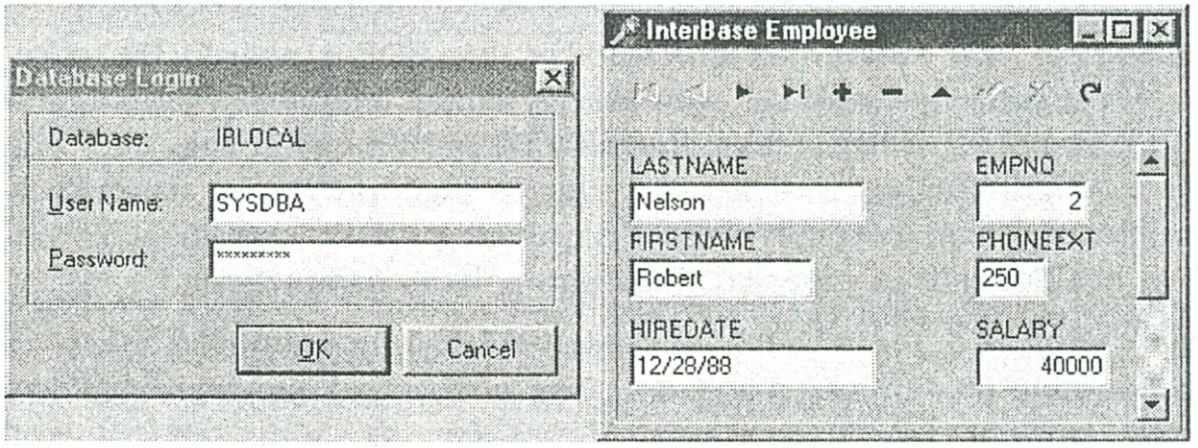
Most lássunk munkához és készítsünk egy helyi InterBase szervert használó egyszerű Delphi alkalmazást. Ezt a programot is a Database Form varázsló segítségével készítjük el, hogy megmutassuk, ez a módszer a kliens/szerver alkalmazások készítésénél is használható (minden amit az előző fejezetben láttunk ugyanúgy működik SQL szerverre is, ahogy ott leírtam).

A program elkészítéséhez készítsünk egy új Delphi projektet, majd válasszuk a **Database | Form Wizard**-ot. Válasszuk a lekérdezésre alapozott egyszerű formot. A következő oldalra lépve válasszuk az IDBLOCAL álnevet (amely a

`Delphi 3.0\Demos\Data` alkönyvtárbeli Mastsql.gdb InterBase fájlra kell mutasson). Ekkor a helyi InterBase szerver egy bejelentkezési párbeszédablakot tár elénk. Ha korábban még nem dolgoztunk a helyi InterBase szerverrel, akkor a beírandó alapértelmezett felhasználói név (login name) „SYSDBA" lesz. Alapértelmezett jelszónak a „masterkey" szót adjuk meg és máris folytathatjuk a Database Form varázsló használatát. Például ha az IbEmp példát akarjuk újra elkészíteni, akkor kapcsolódásnak válasszuk az Employee táblát.

Ebben az esetben a táblák nem fájlok. A fájlválasztó (**List Files of Type**) ablakban a „felhasználói táblák" (**User Tables**) illetve a rendszertáblákat is tartalmazó „minden tábla" (**All Tables**) közül választhatunk. Folytassuk a munkát a varázslóval, jelöljük ki a tábla minkét mezejét és válasszunk egy elhelyezésfajtát (én a függőlegeset választottam, felül címkézve). Ezután készítsük el a formot.

Rendezzük el a formot tetszésünk szerint. Én bekapcsoltam a táblaszerkesztést (a Query komponens RequestLive tulajdonságát True-ra állítva), átrendeztem a komponensekek rajta, átméreteztem és átállítottam a Navigator komponens Flat tulajdonságát True-ra, valamint új nevet és címkét adtam neki. A CD-n is megtalálható, IbEmp névre keresztelt program egy képét a 17.1. ábra mutatja.



17.1. ábra *Az IbEmp program futás közben. Nézzük meg a bejelentkező párbeszédablakot, amely megköveteli a felhasználótól a helyes jelszó megadását, mielőtt a program fő formja megjelenne*

Ez a program minden indításkor megkérdezi a felhasználói nevünket és jelszavunkat (nézzük meg ismét a 17.1. ábrát). Ennek elkerüléséhez hozzáadhatunk egy adatbázis komponenst a formhoz és beállíthatjuk a megfelelő tulajdonságokat. Egy másik megoldás lehet, hogy lekezeljük ezen komponens OnLogin eseményét és beírunk valamilyen értéket a LoginParams tulajdonsághoz. Mindkét lehetőséggel elkerülhetjük a bejelentkezést, a kérdés már csak az, hogy biztosan így akarjuk-e?

Mindig tartsuk szem előtt azt, hogy a bejelentkezési kérelem az egyik kulcsfontosságú vonása az adatbázisok biztonságának. A program fejlesztése alatt hasznos lehet kikapcsolása, de a felhasználóktól általában meg kell követelnünk, hogy bejelentkezzenek mielőtt az alkalmazást használni tudnák. Ebben a példában a helyi SQL adatbázist használtam, amely tehát ugyanazon a gépen volt amelyet a Delphi program fejlesztésére használtam. De nem ez az egyetlen lehetőség. Készíthettem volna ugyanezen lépésekkel egy igazi kliens/szerver alkalmazást, egy hálózaton keresztül az enyémmel összekötött távoli szervert használva.

Kapcsolat távoli SQL szerverrel

A helyi adatbázisok és egy SQL szerver használata között az egyetlen igazi különbség az, hogy az utóbbihoz a Delphi Client/Server Suite változatára, annak is SQL kapcsolataira (SQL Links) van szükségünk. Tulajdonképpen a legtöbb SQL szerverre kapcsolódhatunk ODBC-n keresztül is, de ez a megoldás messze nem a legjobb, ha a hatékonyságot tartjuk szem előtt. Az SQL Links valójában a Borland Database Engine egy kiterjesztése. Ahogyan azt a 16. fejezetben elmondtam (és a 16.1. ábrán látható volt) a BDE több adatforrást, például a Paradox és dBase táblákat közvetlenül elér, másokhoz az ODBC (Open Database Connectivity) illetve az SQL Links csatolófelületét használja.

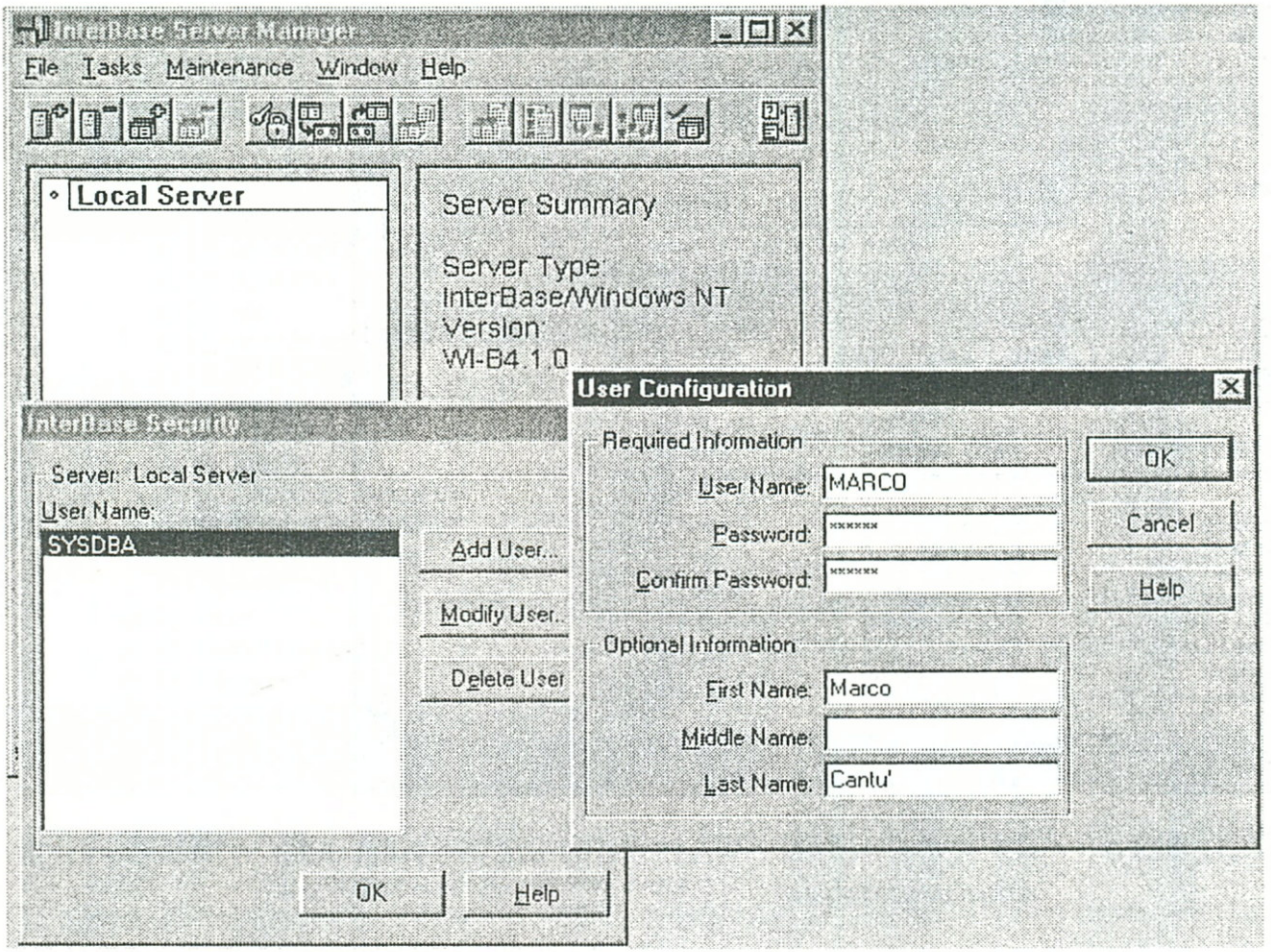
A Delphivel egy SQL szerverhez kapcsolódás akár BDE-n, akár ODBC-n keresztül több rétegen keresztül valósul meg, így ennek telepítése és konfigurálása több lépésből áll. A BDE-t telepíthetjük a Delphi-vel együtt vagy attól függetlenül; konfigurálásához a BDE konfigurációs segédprogramot használhatjuk. Ezzel a programmal új álneveket adhatunk meg, ODBC csatolókat telepíthetünk és beállíthatjuk az egyes adatbáziskapcsolatok paramétereit.

A második réteg az SQL Links-é, amit szintén ebből a programból konfigurálhatunk. E mögött ott van még a hálózati réteg, amihez egy megfelelő hálózati protokollt kell telepítenünk. Utolsó lépésként az SQL szervert kell beállítanunk.

InterBase szerver eszközök

A Delphi olyan alkalmazásokat is tartalmaz, melyekkel mind a helyi InterBase, mind a távoli szerverek beállítását, konfigurálását és üzemeltetését elvégezhetjük. Most röviden bemutatom ezen eszközöket. Működésük teljes leírását megtalálhatjuk a Borland dokumentációi között, ami ezen a téren igen kitejedt.

Szerver-menedzser (Server Manager) – A szerver-menedzser a helyi illetve távoli InterBase szerverek és adatbázisok adminisztrálására használható. Beállíthatók vele a védelmi lehetőségek (új felhasználók létrehozása és törlése, jelszavak meghatározása), készíthetünk biztonsági (backup) másolatokat valamint elvégezhetünk vele különböző üzemeltetési és egyéb feladatokat. A 17.2. ábra egy új felhasználó felvételét mutatja a szerver-menedzser használatával.



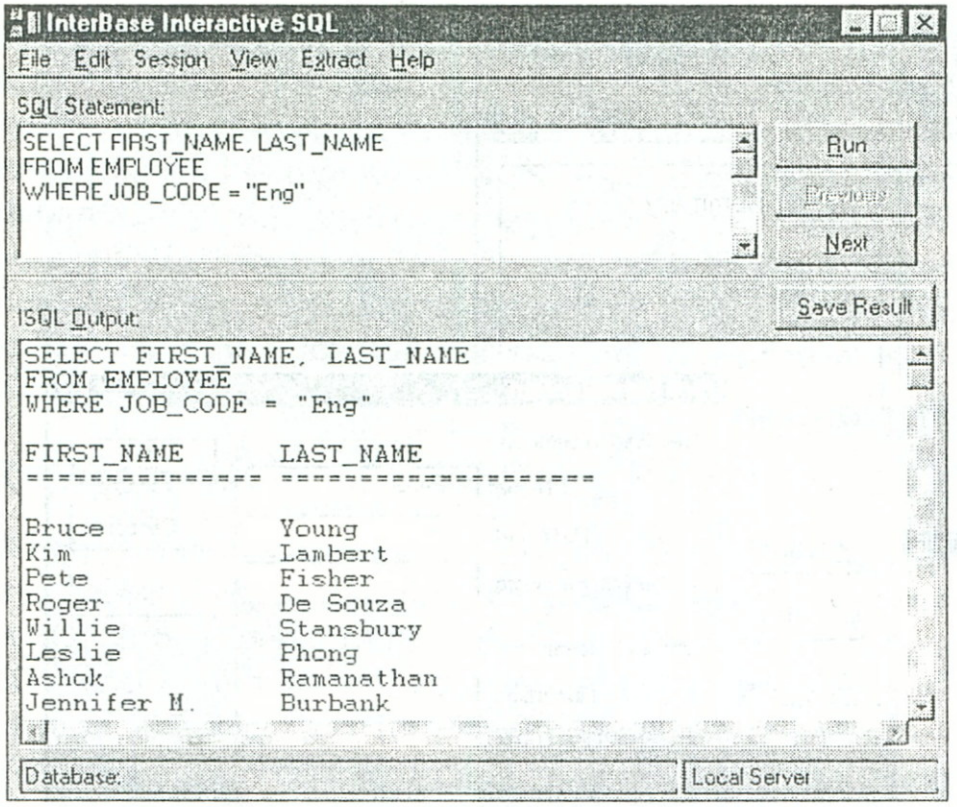
17.2. ábra *Az InterBase Server Manager eszköze, egy felhasználó hozzáadása közben*

Windows ISQL – A Windows InterBase SQL (vagy Interactive SQL) használatával SQL parancsokat hajthatunk végre helyi vagy távoli InterBase szervereken. Az ISQL indítása után hozzákapcsolódhatunk egy már meglevő helyi vagy távoli adatbázishoz és tetszőleges SQL parancsot hajthatunk végre. Például csatlakozhatunk a helyi IBLOCAL álnévhez és végrehajthatjuk az alábbi parancsot:

```
select First_Name, Last_Name
from employee
where Job_Code = "Eng"
```

Ez az SQL parancs kiírja minden tervezőosztályon (Eng) dolgozó munkatárs kereszt- és vezetéknevét, amint azt a 17.3. ábrán láthatjuk.

A Windows ISQL-t tehát használhatjuk tetszőleges adatbázis tartalmának megtekintésére, de igazi jelentőssége az adatbázisok létrehozásában és üzemeltetésében van. Új táblákat, indexeket, (tárolt) eljárásokat stb. hozhatunk létre vele. Ezt mind az InterBase SQL használatával érhetjük el, így általában nem az egyszerű felhasználók használják.

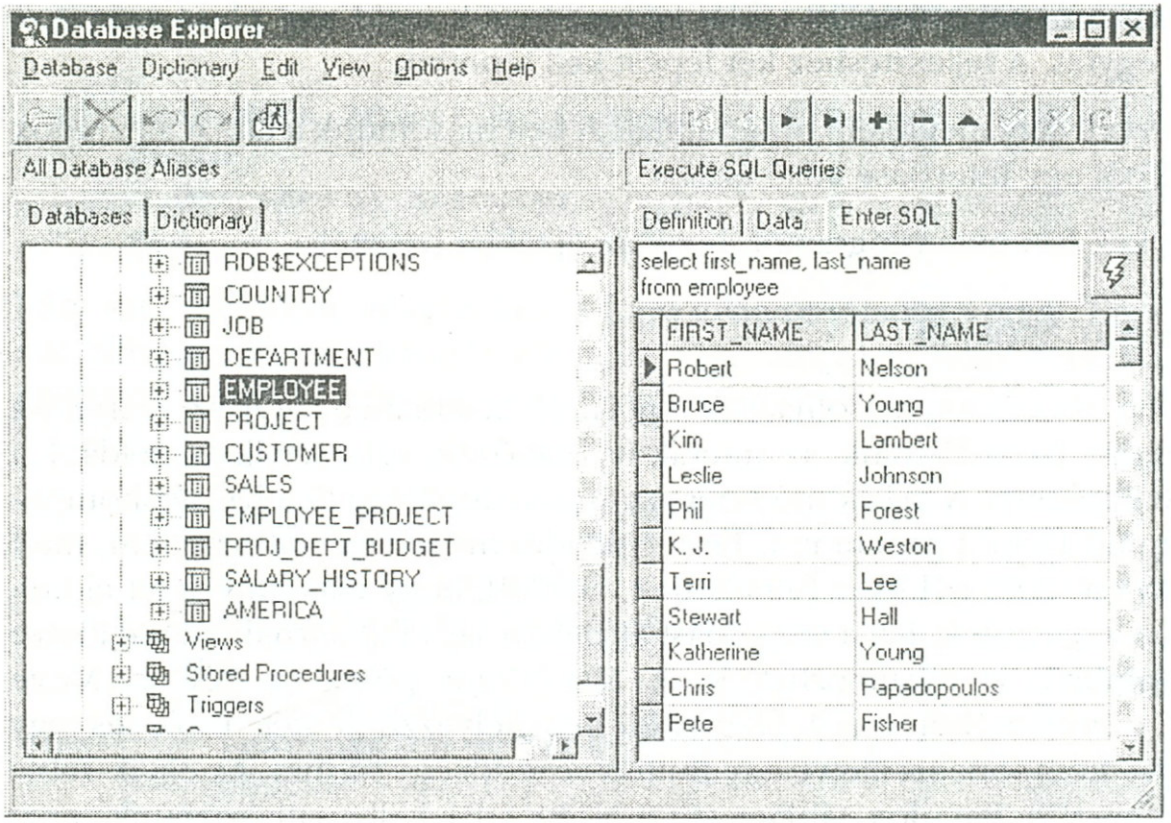


17.3. ábra *Az SQL utasítás végrehajtásának eredménye a Windows ISQL alkalmazásban*

Megjegyzés

Az InterBase szerver által megértett SQL nyelv jóval bővebb annál, mint amit a B függelékben bemutatok illetve a BDE által használt szabvány SQL-nél. Szinte minden SQL szerver tartalmaz olyan lehetőségeket, amelyek nem részei az SQL szabványnak.

Database Explorer és Database Desktop – Egy másik eszköz a helyi vagy távoli InterBase szerverek használatára a Database Explorer. Ezzel lehet a legkönnyebb böngészni a már létező adatbázisok és táblák között, valamint rekordokat törölni, létrehozni illetve módosítani a már meglevőeket. A Database Explorerhez szorosan kapcsolódik a Database Desktop. Tulajdonképpen mindkettő táblanézegető és konfiguráló, míg a Server Manager és a Windows ISQL adatbáziskonfiguráló és létrehozó eszköznek tekinthető. Ezen eszközök között átfedés is van: egy InterBase adatbázisra vonatkozó SQL lekérdezést végrehajthatunk Windows ISQL-ben, Database Explorerben vagy Database Desktopban is (és ekkor még nem említettük a Visual Query Buildert). A Database Explorer használatával például végrehajthatunk egy hasonló SQL parancsot, mint amit az előző Windows ISQL példában láttunk (17.3. ábra). A Database Explorer ilyen használatát a 17.4. ábra mutatja.



17.4. ábra *Egy, az alkalmazottakra vonatkozó SQL lekérdezés eredménye a Database Explorer-ben*

A Data és SQL fülek használatával egyszerűen tekinthetünk meg minden adatot vagy csak néhány mezőt illetve rekordot. Amennyiben többet szeretnénk tudni a Database Explorerről és az InterBase szerver eszközökről, úgy nézzük meg a súgót vagy a kézikönyvet. Mi most néhány Delphi példaprogrammal folytatjuk.

Meglévő program átírása SQL szerverre

Láthattuk, hogy egy szerverre kapcsolódó Delphi adatbáziskezelő program fejlesztése nagyon hasonló a csak helyi fájlokat használóéhoz. Akkor mit kell tennünk, ha egy fájlokkal dolgozó alkalmazást tovább akarunk fejleszteni SQL szervert használóvá? Nem sokat. Egy kicsit több munkával kihasználhatjuk az SQL szerver többi lehetőségét, például a tranzakciókezelést is. Ebben a részben azt nézzük meg, milyen lépéseket kell tennünk, ha egy már kész, fájlokat használó Delphi alkalmazást helyi InterBase szerverre akarunk átültetni. Általában ez az első lépés a távoli SQL szerverre való átállásban. A második lépés ennél sokkal egyszerűbb.

Példaként lássuk azokat a lépéseket, amelyekkel a 16. fejezetbeli Calc példát továbbfejleszthetjük. Az új verzióban, mely már a helyi InterBase szerverre

kapcsolódik, a COUNTRY.DB táblát nézegethetjük majd egy plusz számított mezővel együtt. A fejlesztéshez két lépést kell tennünk:

- Kézzel vagy megfelelő programmal át kell másolnunk a táblát a Paradox fájlból egy InterBase adatbázisba.
- Át kell írnunk a programot, hogy az új táblát használja.

A tábla átmásolása

Alapvetően két módon másolhatunk egy Paradox táblát egy SQL szerverre. Az egyik szerint használhatjuk az interaktív eszközök valamelyikét, például a Database Explorert. A másik módszer szerint írhatunk egy BatchMove komponensre épülő Delphi programot. Én a második megoldást választottam, mert ezt követve a többi példát is futtathatjuk, anélkül, hogy más műveletet el kellene rajtuk végeznünk. Ezt a megoldást általában saját felhasználóinknál illetve programjainknál is alkalmazhatjuk. A MoveCoun példa (a név a „Move Country" rövidítéséből ered) elkészítéséhez helyezzünk két Table és egy BatchMove komponenst illetve egy Button vezérlőt egy formra. Az egyik Table komponenssel a Paradox táblára (COUNTRY.DB) kell csatlakoznunk, amit ezután azonnal megnyithatunk (az Active tulajdonságot kell True-ra állítani). A másik komponenst a céladatbázisra kell beállítanunk, megadva az új tábla nevét. Én nem választhattam a Country-t célnévként, mert már volt ilyen tábla az IBLOCAL adatbázisban. Ezért – és mivel csak amerikai országok szerepelnek a táblában – az Americas nevet választottam. Miután mindkét táblát megadtuk, felhasználhatjuk őket a BatchMove komponens Source és Destination tulajdonságainak beállítására.

Utolsó lépésként a BatchMove komponens Move tulajdonságának kell megfelelő értéket választanunk. Az alapértelmezés batAppend, amely a rekordokat a céltábla végére fűzi. A következő értékek állíthatóak be:

- batUpdate, amely csak az egyező rekordokat frissíti,
- batAppendUpdate, amely egyszerre frissít és hozzáfűz,
- batCopy, amely létrehozza a céltáblát, majd a forrástáblát egy az egyben átmásolja,
- batDelete, amely törli az egyező rekordokat.

Számunkra a batCopy paraméter a megfelelő. Állítsuk ezt be, válasszuk ki a BatchMove komponenst, majd válasszuk az Execute (végrehajtás) parancsot a form SpeedMenu-jéből. A Delphi ekkor létrehozza az új táblát (miután bekérte a bejelentkező információkat, mivel a céltábla egy InterBase adatbázisban van). Így a táblát anélkül át tudtuk másolni, hogy egyáltalán lefordítottuk volna a programot! Másik megoldásként írhatunk egyetlen sor kódot a gomb OnClick eseményére:

```
procedure TForm1.ButtonMoveClick(Sender: TObject);
begin
  BatchMove1.Execute;
end;
```

Mint azt említettem, az igazi kód a form tulajdonságaiban rejlik. Ennek szokásos szöveges formája következik most. Figyeljük meg, hogy két DataSet komponensünk van, viszont egyetlen DataSource sincs.

```
object Form1: TForm1
  Caption = 'Move Countries'
  object ButtonMove: TButton
    Caption = 'Move'
    OnClick = ButtonMoveClick
  end
  object BatchMove1: TBatchMove
    Destination = Table2
    Source = Table1
  end
  object Table1: TTable
    Active = True
    DatabaseName = 'DBDEMOS'
    TableName = 'COUNTRY.DB'
  end
  object Table2: TTable
    DatabaseName = 'IBLOCAL'
    TableName = 'AMERICAS'
  end
end
```

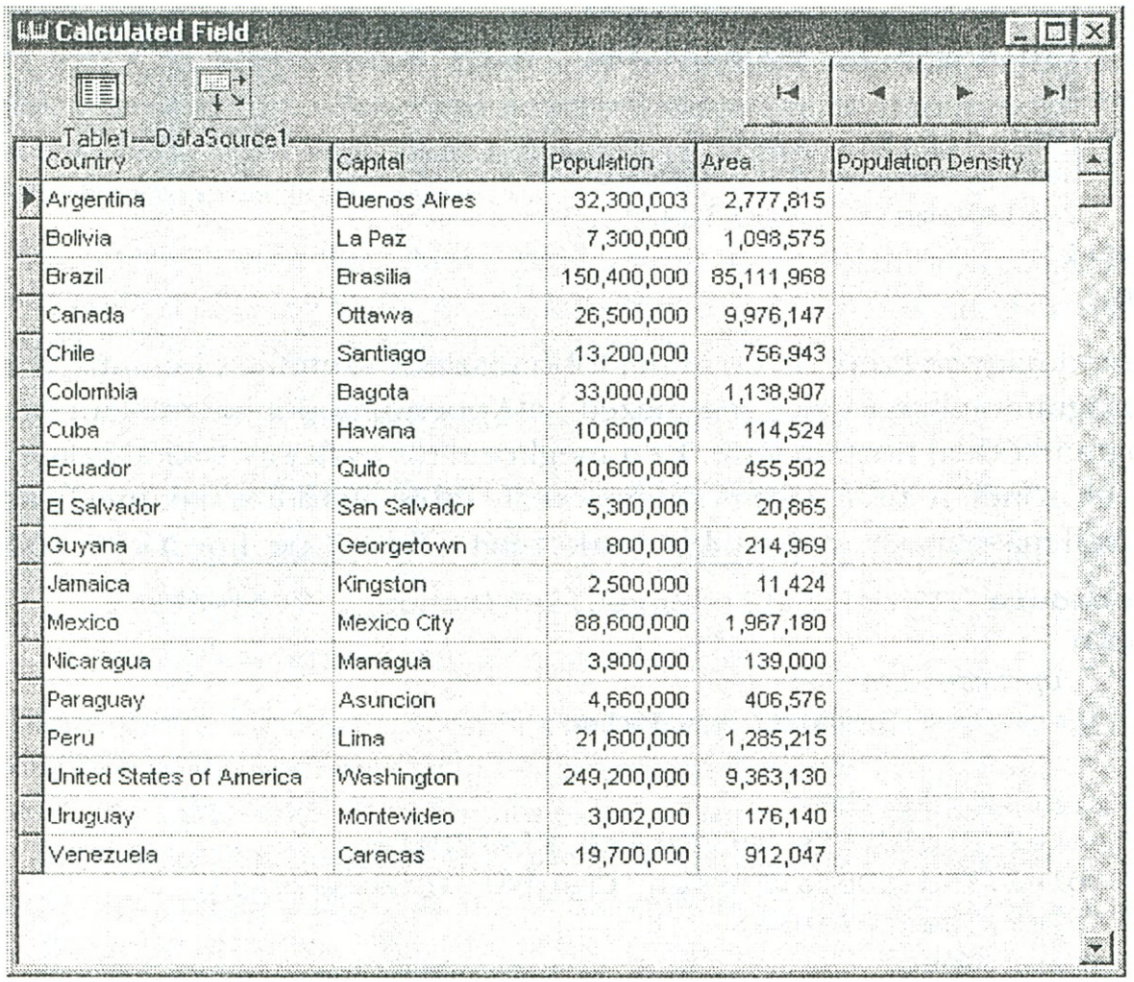
Ez idáig mind nagyon szép, de én ennél több részletet szeretnék bemutatni. E célból a programomban az alapértelmezett batAppend módot használtam és a táblát programkóddal hoztam létre. Ez a megközelítés csak egy fokkal bonyolultabb az előzőnél. A Table2 komponens CreateTable metódusának meghívása előtt át kell másolnunk az első tábla szerkezetét a Table2-be. Íme a kész kód:

```
procedure TForm1.ButtonMoveClick(Sender: TObject);
begin
  // update the UI
  ButtonMove.Enabled := False;

  // copy the data
  Table2.FieldDefs := Table1.FieldDefs;
  Table2.IndexDefs.Assign (Table1.IndexDefs);
  Table2.CreateTable;
  Table2.Open;
  BatchMove1.Execute;
end;
```

A metódus elején letiltjuk a gombot, hogy a felhasználó a program minden végrehajtásakor csak egy másolatot tudjon készíteni. Futtassuk le a programot, kattintsunk a gombra és töltsük ki a megjelenő bejelentkezési párbeszédablakot. Most már készen állunk a továbbfejlesztési művelet második lépésére.

Az alkalmazás átírása

Miután a táblát a helyi InterBase szerverre másoltuk, már foglalkozhatunk a Calc példa fejlesztésével. Az új változatot én Calc2 névre kereszteltem el. Nyissuk meg a régi alkalmazást (vagy egy másolatát), válasszuk ki a Table komponenst, állítsuk az Active tulajdonságot False-ra, majd válaszuk ki az IBLOCAL adatbázist illetve az Americas táblát (ez utóbbinak meg kellett jelennie az Object Inspector-on belül az elérhető táblák listájában). Aktiváljuk ismét a táblát és újra láthatjuk adatainkat, amint azt a 17.5. ábrán is megfigyelhetjük. Csakhogy most már SQL szervert használunk. Amikor lefuttatjuk a programot, megjelenik a számított mező pontosan ugyanúgy, mint az előző változatban. A példa tehát működik, bár teljesítménye a kis méret miatt nem túl lényeges.



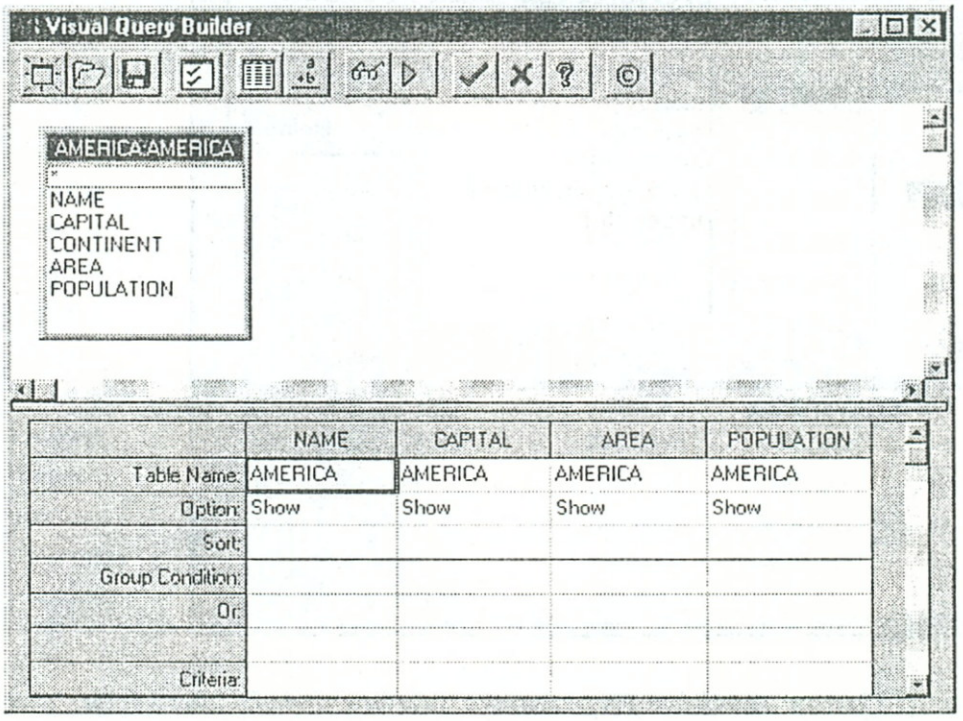
Country	Capital	Population	Area	Population Density
Argentina	Buenos Aires	32,300,003	2,777,815	
Bolivia	La Paz	7,300,000	1,098,575	
Brazil	Brasilia	150,400,000	85,111,968	
Canada	Ottawa	26,500,000	9,976,147	
Chile	Santiago	13,200,000	756,943	
Colombia	Bagota	33,000,000	1,138,907	
Cuba	Havana	10,600,000	114,524	
Ecuador	Quito	10,600,000	455,502	
El Salvador	San Salvador	5,300,000	20,865	
Guyana	Georgetown	800,000	214,969	
Jamaica	Kingston	2,500,000	11,424	
Mexico	Mexico City	88,600,000	1,967,180	
Nicaragua	Managua	3,900,000	139,000	
Paraguay	Asuncion	4,660,000	406,576	
Peru	Lima	21,600,000	1,285,215	
United States of America	Washington	249,200,000	9,363,130	
Uruguay	Montevideo	3,002,000	176,140	
Venezuela	Caracas	19,700,000	912,047	

17.5. ábra *A Calc2 példa formja tervezési időben. Ez a program egy InterBase táblát használ*

A példa következő, Calc3 nevű változata már egy Query (lekérdezés) komponenst használ majd a Table helyett. A formról egyszerűen csak töröljük a táblát, majd adjunk hozzá egy lekérdezést. Kössük össze az adatforrást a lekérdezéssel és írjuk meg a megfelelő SQL parancsot. Mivel mi most a kliens/szerver alkalmazások fejlesztését vizsgáljuk, ezért inkább a próbáljuk meg a Delphi egy másik eszközét, a Visual Query Builder-t használni (alternatív megoldásként egyszerűen beírhatjuk a kívánt SQL parancsot a Query komponens megfelelő tulajdonságához).

A Visual Query Builder használata

A Visual Query Buildert egyszerűen elindíthatjuk, ha kiválasztjuk a Query komponenst, majd a form felett jobb egérgombbal klikkelve előhozzuk a Form Designer helyi SpeedMenu-jét. Válasszuk a Query Builder menüpontot, töltsük ki a bejelentkezési mezőket és válasszunk egy adatbázist. A következő párbeszédablakban csak az Americas táblát jelöljük ki, majd zárjuk be az ablakot. Most már készen állunk a munkára. Amint azt a 17.6. ábrán is láthatjuk, a Visual Query Buildernek van egy alsó területe, ahol a lekérdezés épül, valamint egy felső, ahol a használt táblák láthatóak. A Visual Query Builder eszköztárán található gombokkal készíthetünk új lekérdezést, megnyithatunk vagy elmenthetünk egy már készet, különböző opciókat állíthatunk be, további táblákat nyithatunk meg, számított oszlopokat adhatunk a lekérdezéshez, megnézhetjük a lekérdezés eredményét illetve szövegét, valamint elfogadhatjuk illetve elvethetjük azt.

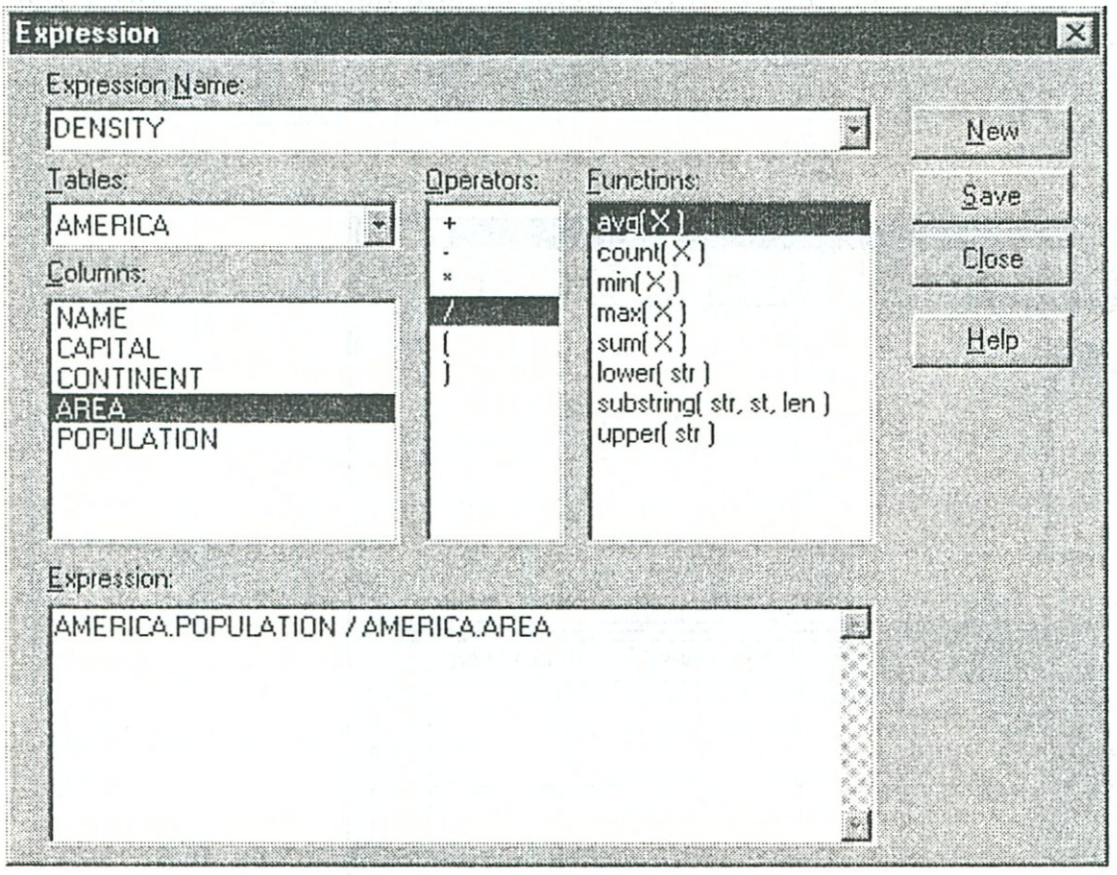


17.6. ábra *A Visual Query Builder főképe, egy táblát és annak néhány mezőjét kijelölve*

Egy lekérdezés elkészítéséhez például húzzunk át néhány mezőt a fenti rész tábláiból az alsó részre, ezzel a lekérdezésbe téve őket. Az alsó részben megadhatjuk a lekérdezés eredménytáblájának rendezési paramétereit, szűrhetjük a kapott rekordokat valamint beállíthatunk sok egyéb hasznos tulajdonságot is. Később látni fogjuk, hogy ha több táblánk szerepel a Visual Query Builderben, akkor grafikusan össze is kapcsolhatjuk őket.

A lekérdezéshez közvetlenül adhatunk számított mezőket is, amint azt a Calc3 példa elkészítésénél látjuk majd. Kattintsunk az eszköztár **Expression** (kifejezés) gombjára. A megjelenő párbeszédablakban adjunk az új kifejezésnek egy nevet (én a Density nevet választottam), kattintsunk duplán a **Population** mezőn, majd az osztásjelen (/), végül az **Area** mezőn.

A kész kifejezés a párbeszédablak alján, az **Expression** részben jelenik meg, amint azt a 17.7. ábra is mutatja. Ha ez szerintünk helyes, akkor már el is menthetjük a lekérdezés szövegét egy fájlba, illetve magát az egész lekérdezést a tábla SQL tulajdonságába a Close gombra kattintva. A lekérdezés eredménytáblájába ekkor egy új mező kerül be. Most már rákattinthatunk a végrehajtás gombra és megnézhetjük az eredményt: a számított mező ugyanúgy ott lesz, mintha csak az eredeti tábla egy mezője lenne.



17.7. ábra *A Visual Query Builder Expression párbeszédablaka számított mezők megadására használható*

Meg is nézhetjük az SQL lekérdezés kódját a szemüveg ikonos gombra kattintva, ami jelenleg a következő:

```
select
  Americas.Name,
  Americas.Capital,
  Americas.Area,
  Americas.Population ,
  ( Americas.Population / Americas.Area ) as Density
from
  Americas Americas
order by
  Americas.Name
```

Ha a pipával ellátott SpeedButton-ra kattintunk, a lekérdezés visszamásolódik a form Query komponensébe. Ha ezt most aktiváljuk, akkor a rácsban meglát-hatjuk a tábla mezőit a számított mezővel együtt, még tervezési időben.

De még mindig van egy kisebb probléma: az SQL kifejezésben megadott osztás művelet eredményeként egy lebegőpontos számot kapunk, elég sok tizedesjeggyel. A korábbi verziókban a számított mező kiszámításánál a Round függvényt használtuk, ami SQL-ben nem használható.

Calculated Field

DataSource1Query1

NAME	CAPITAL	AREA	POPULATION	DENSITY
Argentina	Buenos Aires	2,777,815	32,300,003	11.628
Bolivia	La Paz	1,098,575	7,300,000	6.645
Brazil	Brasilia	8,511,180	150,400,000	17.671
Canada	Ottawa	9,976,147	26,500,000	2.656
Chile	Santiago	756,943	13,200,000	17.439
Colombia	Bogota	1,138,907	33,000,000	28.975
Cuba	Havana	114,524	10,600,000	92.557
Ecuador	Quito	455,502	10,600,000	23.271
El Salvador	San Salvador	20,865	5,300,000	254.014
Guyana	Georgetown	214,969	800,000	3.721
Jamaica	Kingston	11,424	2,500,000	218.838
Mexico	Mexico City	1,967,180	88,600,000	45.039
Nicaragua	Managua	139,000	3,900,000	28.058
Paraguay	Asuncion	406,576	4,660,000	11.462

17.8. ábra *A Calc3 példa formja az SQL által számított mezővel, tervezési időben. Ezt a számított mezőt futási időben is láthatjuk, a Calc2 példával ellentétben (lásd még a 17.5. ábrát)*

A fő gond az, hogy az alapértelmezett rács nem elég széles ahhoz, hogy befogadja a nagy helyiértékű számjegyeket, ezért a megjelenő értékek hibásnak tűnhetnek. A probléma megoldásához meg kell adnunk a lekérdezés mezőire egy-egy megfelelő formátumleíró stringet, a DisplayFormat tulajdonságban. Határoljuk el az ezres értékeket az Area és Population mezőkben (az ###,###,### érték megadásával), majd állítsuk be a Density mezőben megjelenő tizedesjegyek számát (az ###.### értékkel). Ezután megfelelően átméretezhetjük a rács oszlopait, ahogyan a 17.8. ábrán látjuk.

Átírás vagy fejlesztés?

A Calc3 példában két érdekes előnyt láthatunk. Az első és legkézenfekvőbb, hogy most már meg tudunk jeleníteni egy számított mezőt tervezési időben. Ez azért lehetséges, mert ezen számítás elvégzéséhez nincs szükségünk lefordított Pascal kódra, mivel ezt a szerver végzi el, az SQL utasítás végrehajtása közben.

Ez egy kulcsfontosságú pont. Tulajdonképpen a számítások egy részét áthelyeztük a kliens alkalmazásból a szerverre. Ez azt jelenti, hogy számításokat tudunk átvinni a kliens számítógépről a kiszolgálóra, amely általában a nagyobb lekérdezések hatalmas adathalmazát gyorsabban és hatékonyabban le tudja kezelni. Jelenleg még a kliens és a szerver ugyanazon a számítógépen fut, de ez az általános nézőpontot nem befolyásolja. Ezt jelenti az igazi kliens/szerver programozás; az alkalmazás elvégzendő munkájának elosztását a szerver és a kliens között. A kliens elsősorban a felhasználói felülettel, míg a szerver az adatfeldolgozással foglalkozik.

Ezen megközelítés eredményeképpen a Calc3 példaalkalmazás elvileg egy kicsivel gyorsabban fut az előző változatnál, bár ilyen egyszerű esetben a különbség nem érzékelhető. Ez az egyik olyan pont, ahol különbség van aközött, hogy egyszerűen átállítjuk az alkalmazásunkat hogy egy távoli adatbázishoz kapcsolódjon, illetve hogy továbbfejlesztjük azt, hogy kihasználja a kliens/szerver megközelítés előnyeit. Persze más előnyök is léteznek. Az SQL szerver használatával növelhetjük például a megbízhatóságot és adatkonzisztenciát, felgyorsíthatjuk az adatok elérését abban az esetben, amikor több felhasználó egyszerre akarja elérni az adatokat, csak hogy egy pár előnyt említsünk.

Egészét tekintve az adatok áthelyezése és az alkalmazást távoli szervert használóvá átírásának megvalósítása a folyamatnak csak az első – de kétségkívül nagyon fontos – része.

Tábla-összekapcsolások a Visual Query Builderrel

Az előző részben láthattuk, hogyan használhatják ki a Delphi Client/Server Suite felhasználói egy speciális eszköz, a Visual Query Builder előnyeit. Felé-

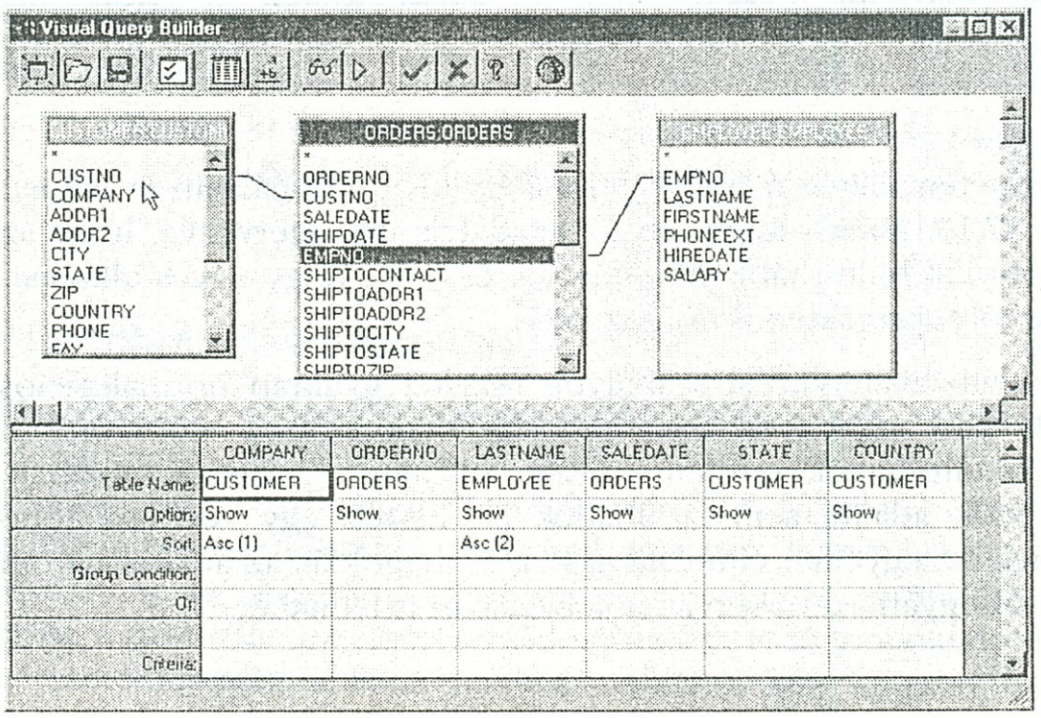
pítettünk vele egy számított mezőt is tartalmazó SQL lekérdezést. Most ennek az eszköznek azt a lehetőségét fogjuk kihasználni, hogy képes különböző táblákat grafikusan összekapcsolni.

Egy háromtáblás összekapcsolás

Mostani, SqlJoin névre keresztelt alkalmazásunk csak három elemből áll: egy Query komponensből, egy adatforrásból és egy kliensterületre igazított rácsból. Alapvetően az egyes projekteken dolgozó alkalmazottak listáját szeretnénk megjeleníteni, az IBLOCAL adatbázisban szereplő adatok felhasználásával.

Ebben az adatbázisban van egy Order (rendelés) tábla, mely több másik táblával áll relációs kapcsolatban, például az Employee (alkalmazott) táblával (hogy megmondhassuk ki vette fel a rendelést) illetve a Customer (vevő) táblával (az előző fejezet végén is hasonló táblákkal dolgoztunk, bár akkor azok még csak lokálisak voltak).

Ahhoz, hogy valamilyen olvasható információt kapjunk, előbb össze kell kapcsolnunk ezt a táblát a Customer táblával – amely a vevők neveit és címeit tartalmazza –, valamint az Employee táblával, amely az egyes alkalmazottakról szóló információkat tartalmazza. A Visual Query Builder segítségével egyszerűen megadhatjuk ezt a két kapcsolatot, csak ki kell jelölnünk a táblákat és a megfelelő mezőket át kell húznunk a másik tábla ide vonatkozó mezőjére. Ezután ki kell jelölnünk a lekérdezés mezőit. Az általam ezzel az eszközzel készített lekérdezést a 17.9. ábrán láthatjuk.



17.9. ábra *Az SqlJoin példa lekérdezése a Visual Query Builder-ben*

Íme a lekérdezést megvalósító SQL parancs:

```
select
  Customer.Company, Orders.Orderno, Employee.Lastname,
  Orders.Saledate, Customer.State, Customer.Country
from
  Employee Employee, Customer Customer, Orders Orders
where
  (Customer.Custno = Orders.Custno) and
  (Employee.Empno = Orders.Empno)
order by
  Customer.Company, Employee.Lastname
```

Valójában csak két tábla mezőit használjuk, a harmadik a két összekapcsolás megfelelő kódok alapján való elvégzéséhez kell.

Figyelem

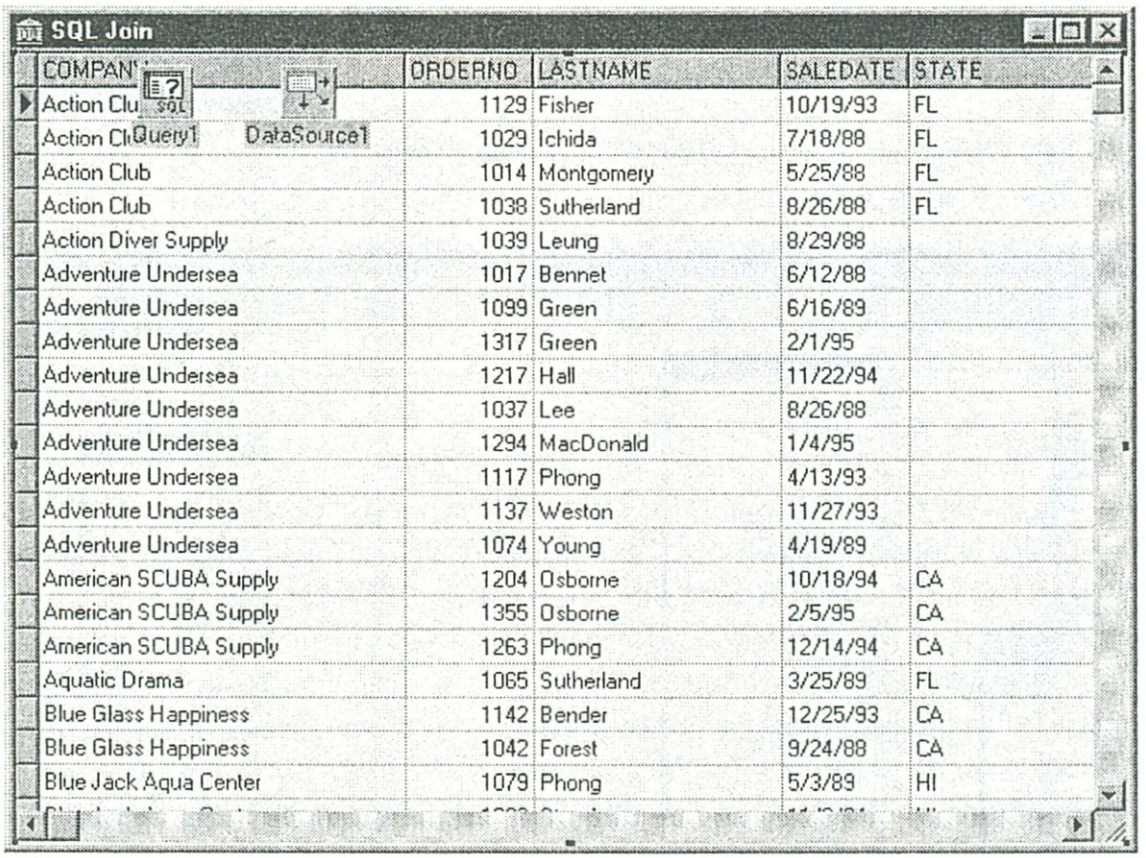
Ha a Visual Query Builder segítségével egy lekérdezést készítünk, akkor annak szövege ténylegesen megjelenik a megfelelő Query objektum SQL tulajdonságánál. De ha ott módosítunk rajta, akkor már többé nem férünk hozzá a Query Builder-beli lekérdezés szerkezetéhez. Ennek elkerülésére azt javaslom, mindig mentsük el a lekérdezést egy fájlba, mielőtt kilépnénk a Visual Query Builderből, hogy később vissza tudjuk állítani azt.

A program kimenetén illetve a táblák fejlesztési idejű vizsgálatán is látható (nézzük meg a 17.10. ábrát), miért lett az adatbázis úgy szervezve, hogy az egyes projektadatokat külön táblákban tárolja. Látható, hogy egyes alkalmazottak egyszerre több projekten is dolgoznak.

A szabványos adatbázis-tervezési szabályok (ezeket gyakran normalizációs szabályoknak hívják) szerint az információkat nem szabad többszörösen tárolni. Ezért nem tárolhatjuk az alkalmazottak adatait a rendelések táblában. Hasonlóan a vevők adatait sem tárolhatjuk itt, hiszen egy vevőnek több rendelése is lehet. Az egyetlen elméletileg helyes megoldás az adatok három táblára bontása, ahogyan azt ebben az adatbázisban is láthatjuk.

Megjegyzés

Ha nem tudjuk, mit értünk adatbázis normalizációs szabályokon, akkor azt javaslom foglalkozzunk adatbázistervezéssel még mielőtt jobban beleásnánk magunkat a komplex adatbáziskezelő programok fejlesztésébe – akár Delphiben tesszük azt, akár nem. Rengeteg könyv szól erről a témáról; közülük sok egyetemi adatbáziselméleti kurzusra íródott.



COMPAN	ORDERNO	LASTNAME	SALEDATE	STATE
Action Clu	1129	Fisher	10/19/93	FL
Action Cl	1029	Ichida	7/18/88	FL
Action Club	1014	Montgomery	5/25/88	FL
Action Club	1038	Sutherland	8/26/88	FL
Action Diver Supply	1039	Leung	8/29/88	
Adventure Undersea	1017	Bennet	6/12/88	
Adventure Undersea	1099	Green	6/16/89	
Adventure Undersea	1317	Green	2/1/95	
Adventure Undersea	1217	Hall	11/22/94	
Adventure Undersea	1037	Lee	8/26/88	
Adventure Undersea	1294	MacDonald	1/4/95	
Adventure Undersea	1117	Phong	4/13/93	
Adventure Undersea	1137	Weston	11/27/93	
Adventure Undersea	1074	Young	4/19/89	
American SCUBA Supply	1204	Osborne	10/18/94	CA
American SCUBA Supply	1355	Osborne	2/5/95	CA
American SCUBA Supply	1263	Phong	12/14/94	CA
Aquatic Drama	1065	Sutherland	3/25/89	FL
Blue Glass Happiness	1142	Bender	12/25/93	CA
Blue Glass Happiness	1042	Forest	9/24/88	CA
Blue Jack Aqua Center	1079	Phong	5/3/89	HI

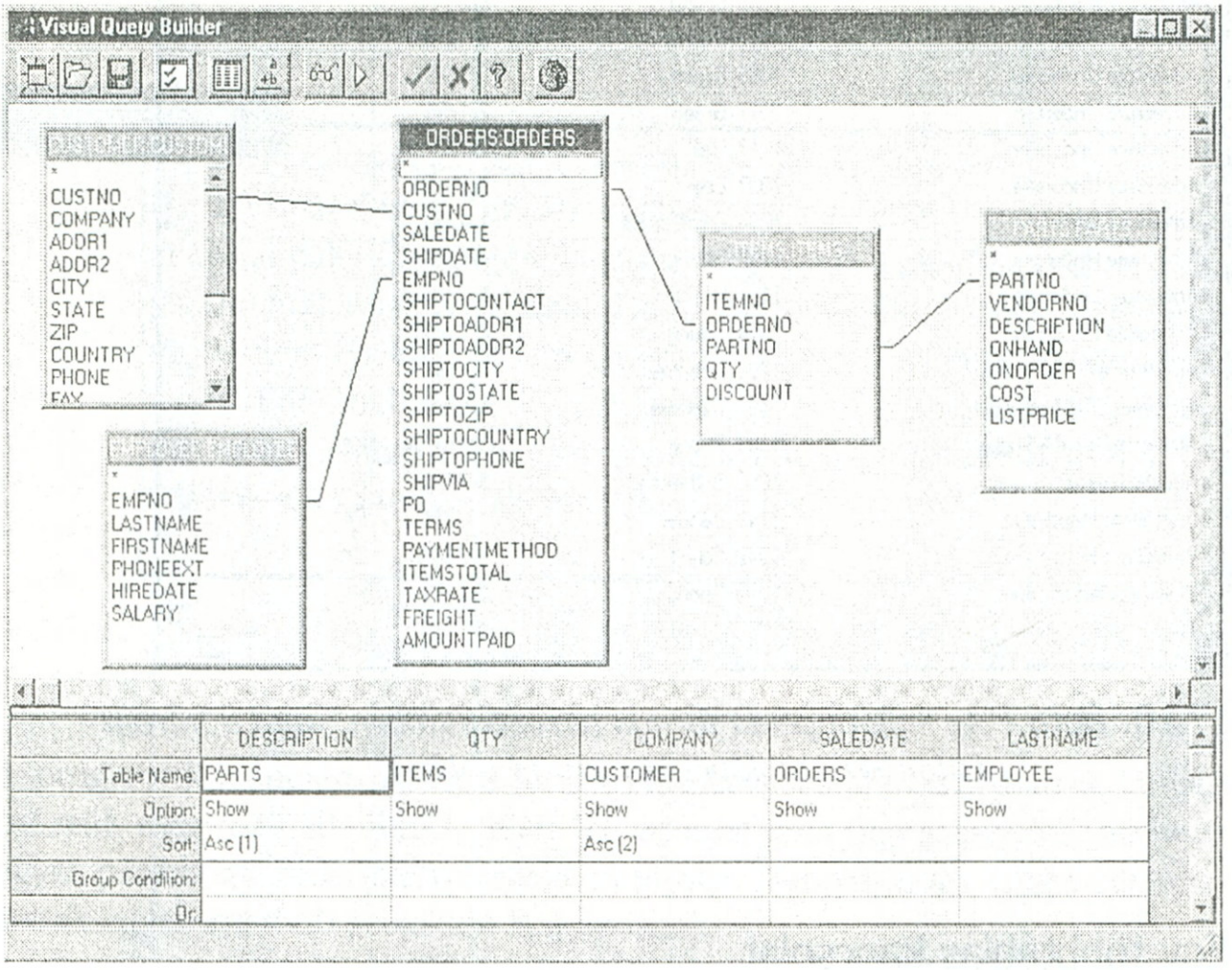
17.10. ábra *Az SqlJoin példa formja tervezési időben, rajta a három különböző táblából kinyert adatokkal*

Egy többtáblás kapcsolat

Programunk már most is érdekes példája három tábla összekapcsolásának, de még mindig van olyan adat, amelyet nem tartalmaz. Például megjeleníthetjük a termékekre vonatkozó eladási információkat. Mivel minden rendelés több tételből állhat, ezért hozzá kell adnunk az Items (tételek) táblát az összekapcsolásunkhoz, így megmondhatjuk mely vevő mely árukat rendelte meg. Az Items tábla viszont csak az egyes rendelésekhez tartozó termékkódokat tartalmazza, a terméknéveket a Parts (cikkek) táblából kell kinyernünk. Végezetül öt

összekapcsolt táblához jutunk, ahogyan azt a 17.11. ábrán láthatjuk. Az alábbi lesz a megfelelő SQL utasítás:

```
select
  Parts.Description, Items.Qty, Customer.Company,
  Orders.Saledate, Employee.Lastname
from
  Orders Orders, Customer Customer, Parts Parts,
  Items Items, Employee Employee
where
  (Customer.Custno = Orders.Custno) and
  (Employee.Empno = Orders.Empno) and
  (Items.Partno = Parts.Partno) and
  (Items.Orderno = Orders.Orderno)
order by
  Parts.Description, Customer.Company
```



17.11. ábra *Az SqlJoin2 példa lekérdezése a Visual Query Builderben*

Adatmodulok

A kliens/szerver programozás egy nagyon érdekes formája az alkalmazásfelosztásnak, ahol is a kódot illetve a munkát kell elosztani két különböző logikai szint – és legtöbbször két különböző számítógép – között. Egyik oldalon ott vannak az adatbázisban tárolt adatok, a másikon egy ezeket elérő Delphi program. Az elmúlt néhány évben egy új, még jobb felosztást megvalósító megközelítés fejlődött ki, a háromrétegű megközelítés. Egyik oldalon vannak az adatok, másikon a program felhasználói felülete, végül a harmadikon az adatok elérését és módosítását meghatározó szabályok és feltételek állnak.

Megjegyzés

A háromrétegű modell a kliens/szerver programozás elég átfogó technikája, ezért itt most nem tudom részleteit és előnyeit kifejteni, csak a fő részeit mutatom be. Ismétlem, az adatbázisfejlesztés általános leírása e könyv keretein bőven túlmutat.

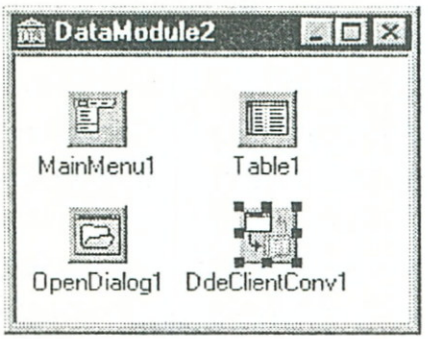
A harmadik, közbülső réteghez sorolhatjuk az adatelérési komponenseket (Table, Query, Data Access stb.). Ezek a nemvizuális komponensek nem tartoznak a felhasználói felülethez, de használhatóak a mezőtulajdonságok beállításban, számított mezők hozzáadásában, a táblákba küldendő új adatok ellenőrzésében stb. Ezeket a komponenseket használva szabályokat építhetünk fel adataink elérésére, bár néhány ilyen szabályt maga az adatbázis is meghatároz. Eddig megírt Delphi programjainkban a vizuális komponensek a hagyományos kétrétegű modellt követve az adatelérő komponensekkel voltak összekötve. A háromrétegű modell megvalósításához az adatelérő komponenseket a megjelenítő komponensektől független, másik formon is elhelyezhetjük, majd ezután egyik formról a másikra hivatkozunk (a **File | Use Unit** használatával). Ezzel a módszerrel tervezési időben összeköthetjük adatfüggő komponenseinket a megfelelő táblákkal.

Ezzel a megoldással az a probléma, hogy olyan formokon helyezzük el az adatelérő komponenseinket, amelyet aztán sohasem mutatunk meg. A Delphi megoldása ezen rejtett formok kezelésére az adatmodulok használata (amit röviden tehát egyfajta rejtett, nemvizuális komponenseket tartalmazó formnak tekinthetünk). Az adatmodulok egyaránt tartalmazhatják az adatelérő komponenseket és az azok eseményeikhez tartozó metódusokat, ezáltal elválasztva az adatbáziskapcsolatot a program felhasználói felületétől.

A Delphi 3-ban az adatmodulok a háromrétegű alkalmazások fejlesztésének építőkövei. Valójában lokálisan is végrehajthatóak (ahogyan az elkövetkező példákban látni fogjuk), de beágyazhatjuk őket a szervergépen végrehajtandó objektumokba is. Ahhoz, hogy a kliens futtassa a felhasználói felület formjait, egy másik szerver az adatmodult, és egy nagyobb szerver foglalkozzon az adatbázissal, egy fizikailag is háromrétegű alkalmazást kell készítenünk. Ezt a Delphi 3 támogatja, de ez már túlmutat jelen könyv keretein.

Egy adatmodul elkészítése

Az adatmodul elkészítéséhez készítsünk egy új, alapértelmezés szerinti alkalmazást, majd adjunk hozzá egy adatmodult a **File | New Data Module** kiválasztásával (ha a Database Form varázslót használjuk, akkor használhatjuk a megfelelő opciókat az utolsó oldalon és így egyszerre készíthetjük el a formot és az annak adatelérő komponenseit tartalmazó adatmodult). Az adatmodul egy új, üres fehér ablakként jelenik meg, amihez a szokásos módon adhatunk komponenseket. Két fő különbség van: először is nem adhatunk vezérlőket az adatmodulhoz, másrészt az adatmodul-ablak csak a modul komponenseinek fejlesztési idejében látható, ahogyan azt a 17.12. ábra mutatja.



17.12. ábra *Egy különböző nemvizuális komponenseket tartalmazó adatmodul*

A TDataModule osztály közvetlenül a TComponent-ből származik, tehát gyakorlatilag semmi köze a Windows ablakfogalmához. A form és az adatmodul közötti látszólagos hasonlóság mindjárt szembetűnőbb, ha észrevesszük, mennyivel kevesebb tulajdonsága és eseménye van ez utóbbinak. Éppen ezért egyszerűbb az adatmodulokat memóriában tárolt komponens és metódustárolóknak tekinteni.

Vajon mi a közös az adatmodulokban és a formokban? Miért hasonlítanak annyira egymásra a Delphi fejlesztők szemében? A formokhoz hasonlóan az adatmodulok is egyes Object Pascal unitokban tárolják a definícióikat és form-

leíró (DFM) fájlokban a modulhoz tartozó egyes komponenseiket illetve azok tulajdonságait:

```
object DataModule2: TDataModule2
  Height = 159
  Width = 196
  object Table1: TTable...
  object DataSource1: TDataSource...
end
```

Egy Delphi adatmodul egységes struktúrája nagyon hasonló a formokéhoz. A legfontosabb különbség a szülőosztálynál van:

```
type
  TDataModule2 = class(TDataModule)
```

A másik hasonlóság a formok és adatmodulok között az, hogy mindkettő elkészíthető amikor az alkalmazást elindítjuk vagy esetleg később. Tulajdonképpen az adatmodulok még a **Project Options** oldal **Forms** felsorolásában is szerepelnek.

Megjegyzés

Azt gondolhatnánk, hogy az adatmodulok az adatbáziskezelő alkalmazásoknak csak a fejlesztéséhez tartoznak hozzá. Ez általában így van, de a szeparált kódmoduloknak máshol is hasznát vehetjük. Erre példa lehet a rendszerkomponensek, például a DDE komponensek kezelése. Valahányszor egy olyan komponenssel kell dolgoznunk, amely nem tartozik szigorúan egy formhoz vagy több formon is felhasználható, használhatunk másodlagos adatmodulokat. Ez az út hasznos lehet például akkor, amikor többszörös öröklődést szeretnénk utánozni (tehát amikor más-más modulban megvalósított különböző képességeket szeretnénk egy formhoz adni).

Adatmodul több nézethez

Az adatmodulok egy tipikus alkalmazása az, amikor ugyanannak az adatnak több nézetét szeretnénk láthatóvá tenni és szeretnénk, ha ezek a nézetek szinkronban maradnának. Ezt oldottam meg a TwoViews példában. Később majd ezt a példát terjesztem ki szabályok és szűrők hozzávételével. A TwoViews példához két formot és egy adatmodult készítetem. Ezután beállítottam a DBDEMOS adatbázis CUSTOMER.DB tábláját és az adatforrást. Ennek az alkalmazásnak nagyon egyszerű az adatmodulja:

```
object DataModule2: TDataModule2
  object Table1: TTable
    Active = True
    DatabaseName = 'DBDEMOS'
    TableName = 'CUSTOMER.DB'
  end
  object DataSource1: TDataSource
    DataSet = Table1
  end
end
```

Tipp

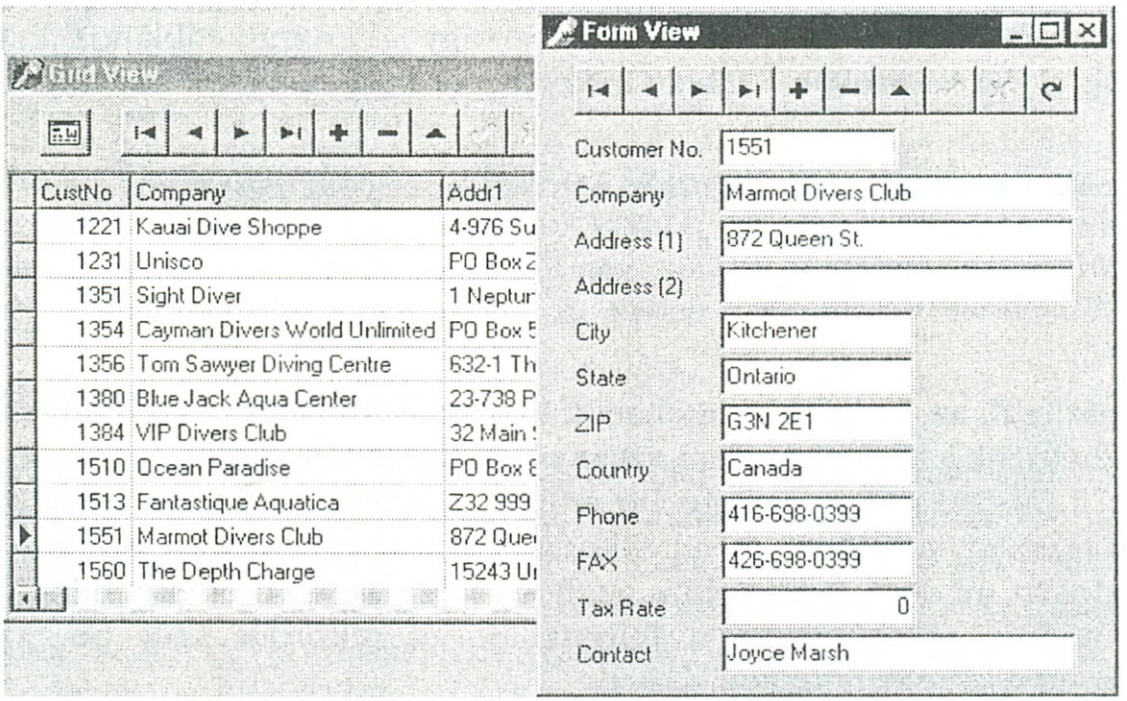
Mielőtt még elkezdenénk használni egy új unitot a formon, adjunk rendes nevet neki. Valójában amikor egy új (például Unit2 nevű) unitot használunk, majd elmentéskor más nevet adunk neki, akkor a kapcsolatunk elveszik vele, és kézzel kell majd az összes rá való hivatkozást átírnunk még akkor is, ha a `uses` utasításokat a Delphi írta meg nekünk a **File | Use Unit** parancs hatására.

Ezután készítettem egy eszköztárat a program fő formjához, felülre igazított panellel, rajta egy gombbal és egy DBNavigator-ral. A speedbuttont használom a második form megjelenítésére. A form többi részét egy DBGrid foglalja el. A form adatmoduljának felhasználásával beállíthatjuk mind a DBNavigator mind a DBGrid DataSource tulajdonságát DataModule2.DataSource1-re.

A második nézet egy olyan form, amelyen több DBEdit komponens van, az utolsó kivételével a tábla minden mezőjéhez egy. Ahelyett, hogy minden DBEdit komponenst egyenként elhelyeznénk a formon majd egyenként kapcsolódnánk velük, jelöljük ki az adatmodul tábla komponensét, nyissuk meg a mezőszerkesztőt, adjuk hozzá az összes mezőt, jelöljük ki mindet az utolsó kivételével, majd húzzuk át a kijelölt mezőket a második formra. Ezután az egyszerű művelet után a Delphi rögtön elrendezi nekünk a megfelelő DBEdit és Label komponenseket a formon. Most beállíthatjuk a második form Visible tulajdonságát True-ra, így az azonnal megjelenik, amint a programot elindítjuk (17.13. ábra).

Ha mindkét formot megjelenítjük, akkor láthatjuk, hogy azok szinkronban vannak. Az egyiken levő navigációs gombokkal mindkét formot vezérelni tudjuk. Valójában a navigátor egyikhez sem tartozik; az adatmodul adatforrásához van kötve, a vizuális komponenseket pedig a közös adatelérő komponens

egyes változásai befolyásolják. Ha az egyik formon változtatunk, akkor a másik is azonnal megváltozik, amint a változtatásokat jóváhagyjuk. Ha új rekordot viszünk fel, az mindkét formon látható lesz.



17.13. ábra *A TwoViews program futási időben, amint a két szinkronizált form ugyanazt a rekordot ábrázolja*

Figyeljük meg, hogy a rekordok között tervezési időben is böngészhetünk. Amikor a rácsot arrébbgörgetjük, a második formon is megváltoznak az adatok, így lehetővé válik például, hogy átméretezzünk egy DBEdit komponenst, ha az aktuális rekord valamelyik mezője túl hosszú és ezért nem fér el a megadott helyen.

Mezőtulajdonságok és kezdőértékek megadása

Az adatmodulok használata két form szinkronban tartására elég egyszerű és hasznos megoldás. Most szeretnénk a programhoz további olyan képességeket adni, amelyek magukhoz az adatokhoz tartoznak, nem pedig azok megjelenítéséhez. Például módosíthatjuk az adatmodulban a mezők tulajdonságait, valamilyen speciális értéket megadva a Table1Phone és Table1FAX mezőkomponensek EditMask tulajdonságához. Ezek a beállítások mindkét formon egyszerre érvényesülnek majd az adott mezők nézegetésekor és szerkesztésekor.

Ahhoz hogy valami bonyolultabbat érjünk el, bevezethetünk például egy új szabályt a táblára, vagy legalábbis egy új lehetőséget a felhasználók számára. Szeretnénk minden egyes új vevőhöz automatikusan létrehozni egy egyedi

vevőkódot, amely az ebben a mezőben eddigi legnagyobbnál egyel nagyobb kell legyen. Én ezt egy apró kódrészlet hozzáadásával értem el, melyet a program második verziójának – a TwoView2-nek – adatmoduljához adtam.

Alapvetően mindig a Table1CustNo mezőnek kell megfelelő értéket adnunk, valahányszor a felhasználó új rekordot szúr be vagy ad hozzá a táblához. Ezt úgy érhetjük el, ha lekezeljük a tábla OnNewRecord eseményét a következő módon:

```
procedure TDataModule2.Table1NewRecord(DataSet:
TDataSet);
begin
  Table1CustNo.Value := Max + 1;
end;
```

A kérdés már csak az, hogyan számíthatjuk ki a Max értéket? Egyszerűen végigböngészhetnénk a táblát, ahogyan azt az előző fejezetben tettük, és megkereshetnénk a CustNo mező legnagyobb értékét. Csakhogy ezt nem tehetjük meg a fenti eseménykezelőből, mivel ekkor az visszaállítaná a táblát tsBrowse módba a tsInsert módból. A megoldás az, hogy a legnagyobb értéket mindig újra kiszámoljuk, valahányszor a felhasználó új rekordot szúr be, az OnBeforeInsert esemény felhasználásával:

```
procedure TDataModule2.Table1BeforeInsert(DataSet:
                                                TDataSet);
begin
  ComputeMax;
end;
```

A ComputeMax eljárás egyszerűen végigböngészi a táblát és megkeresi a legnagyobb értéket. Itt következik ennek a metódusnak a kódja, amely az esetleges hibákat is kezeli egy `try-finally` blokk használatával, könyvjelzőket használ és szükség szerint letiltja/engedélyezi a vezérlőket:

```
procedure TDataModule2.ComputeMax;
var
  Bookmark: TBookmark;
begin
  Bookmark := Table1.GetBookmark;
  try
    Table1.DisableControls;
    Max := 0;
    try
      Table1.First;
      while not Table1.EOF do
      begin
        if Table1CustNo.AsInteger > Max then
          Max := Table1CustNo.AsInteger;
        Table1.Next;
      end;
```

```
      finally
        Table1.EnableControls;
      end;
    finally
      Table1.GotoBookmark (Bookmark);
      Table1.FreeBookmark (Bookmark);
    end;
  end;
```

Amint újabb metódusokat adunk az adatmodulhoz, folyamatosan a háromrétegű alkalmazásmodell felé lépünk tovább. Ez a kód valójában teljesen független a felhasználói felülettől (a két nézettől). A példában szereplő kód nagyon egyszerű, mégis jól megvilágítja ezt a fontos ötletet.

Egyszerű táblaszűrés

Most szeretném a példaalkalmazásban megvalósítani (ismét csak az adatmodul használatával) az adatok mindkét formban történő szűrését. Erre a legegyszerűbb lehetőség a Delphiben egy indexelt mező lehetséges értékkészletének meghatározása. Én például a TwoView2 tábláját a ByCompany másodlagos index segítségével rendeztem (csak be kell állítanunk ezt az értéket az IndexName tulajdonságban). Ezután kiválasztottam minden olyan rekordot, amely két, a felhasználó által megadott érték közé esik, a következő sor használatával:

```
Table1.SetRange (['Abacus'], ['Custom']);
```

Alternatív megoldásként csakúgy, mint a GotoKey metódus használatánál, most is beállíthatunk két kulcsértéket a SetRangeStart illetve SetRangeEnd metódusokkal, majd ezután meghívhatjuk az ApplyRange metódust. Általában egyszerűbb a SetRange-et meghívni változók két, az aktuális indexelés mezőivel azonos hosszúságú (és azonos rendezésű) tömbbel. Amikor már nem akarjuk tovább használni a tartományszűrést, hívjuk meg egyszerűen a CancelRange metódust.

A TwoView2 programban valójában nem rögzített tartományt adtam meg, mint az fentebb említettem, hanem készítettem egy új párbeszédablakot, amelyben a felhasználó megadhatja a tartomány kezdő és végértékét. Használhattam volna a fő form eszköztárát is az új párbeszédablak helyett, de ismét csak szerettem volna a tartománybeállító kódot az adatmodulhoz és nem valamely adatokat megjelenítő formhoz kapcsolni. Tehát készítettem egy formot egy jelölődobozzal és két szerkesztőmezővel, hozzáadtam egy új, saját metódust az adatmodulhoz, amelynek a következő lett a kódja:

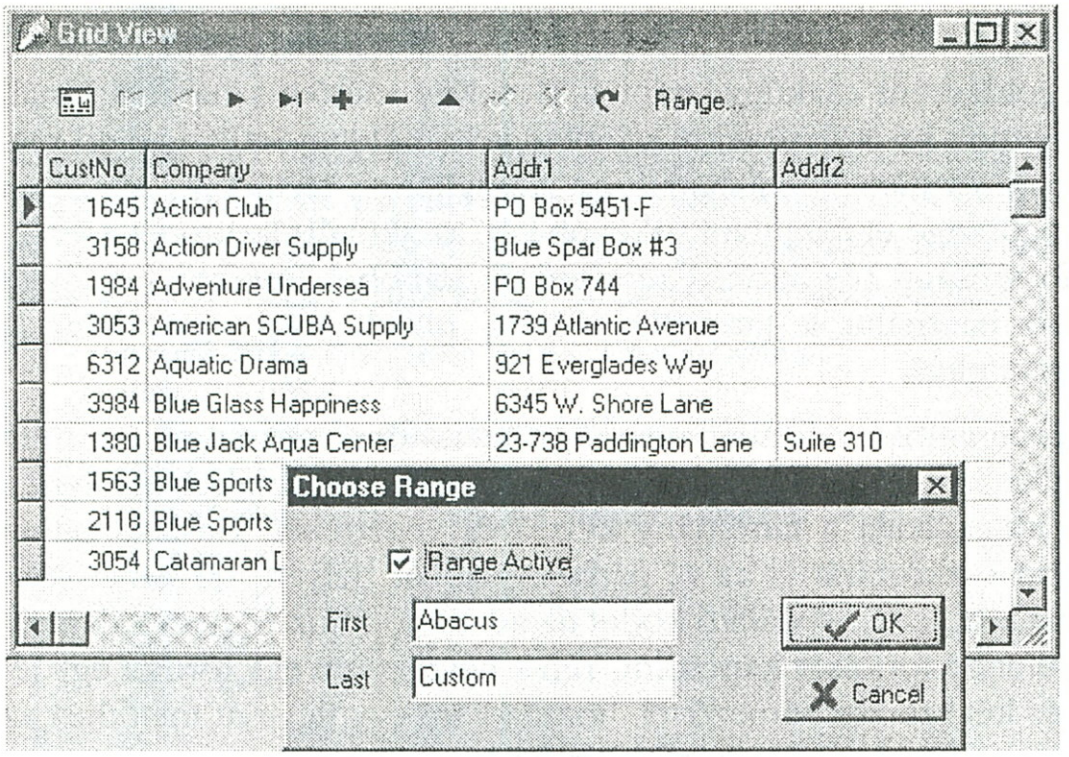
```
procedure TDataModule2.ChooseRange;
begin
  with FormRange do
    if ShowModal = mrOK then
```

```
        if CheckBoxRange.Checked then
          Table1.SetRange ([Edit1.Text], [Edit2.Text])
        else
          Table1.CancelRange;
    end;
```

Látjuk, hogy az adatmodul nem tartalmaz vizuális komponenseket, de az input/output megvalósításának céljából támaszkodhat más formokra illetve párbeszédablakokra. Most már minden form meghívhatja ezt a metódust, ahogyan ezt a fő form eszköztárának új, Range gombjára kattintáskor meg is tesszük:

```
procedure TForm1.RangeSpeedButtonClick(Sender: TObject);
begin
  DataModule2.ChooseRange;
end;
```

Az új szűrés használatának hatását és az új párbeszédablakot a 17.14. ábrán tekinthetjük meg. Ennek az ablaknak a használata egy újabb lehetőség az adatok és a formok közötti közreműködés testreszabásának, az adatelérési réteg használatával. Kétségtelenül, hogy használunk felhasználói felületet is az adatmodulban, de észrevehető, hogy ez a párbeszédablak nem mutat semmilyen adatbázisbeli adatot, csak megjelenítési lehetőségeket. Emiatt nyugodtan tekinthetjük a harmadik réteg, az adatmodul részének.



17.14. ábra *A tartomány meghatározására használt párbeszédablak és annak hatása a rácsra*

Egyéni táblaszűrés

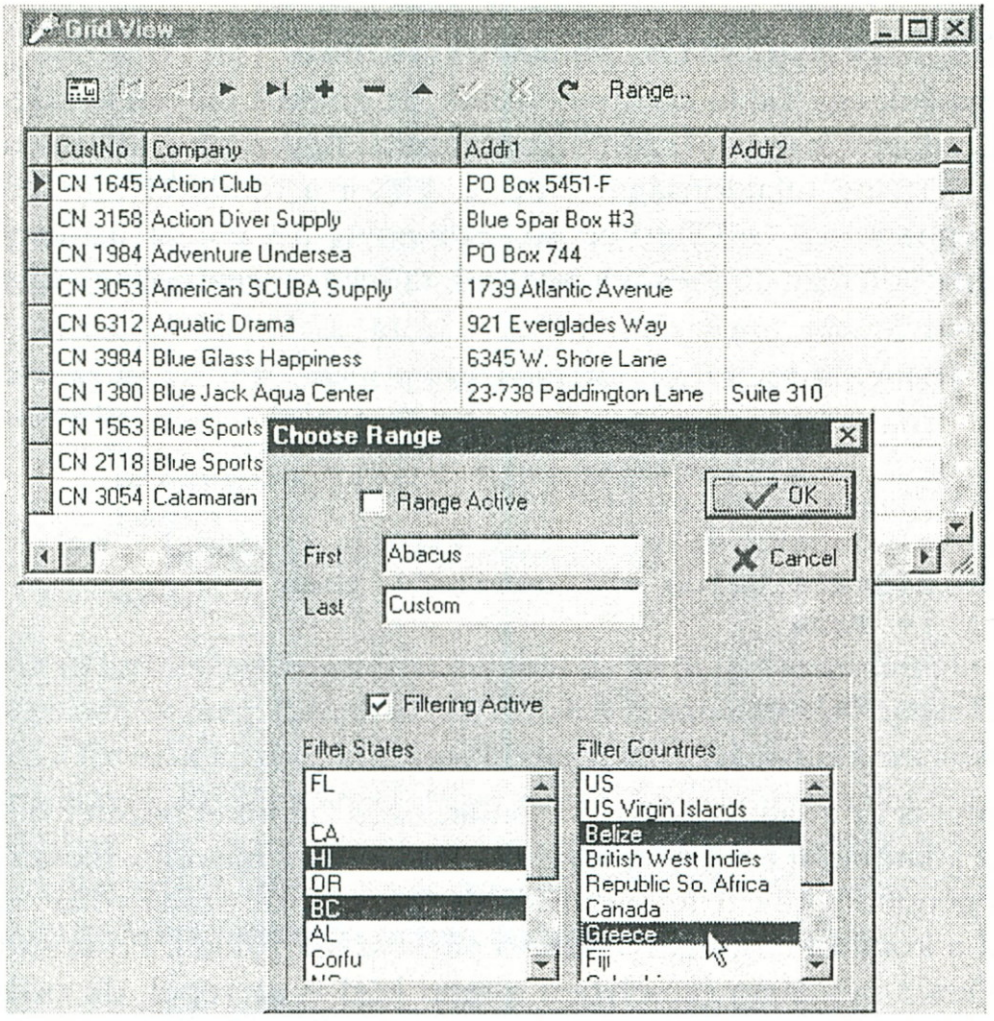
Amellett, hogy megadjuk egy Table komponensnek, az értékek milyen készletén dolgozzon, megadhatunk egyéni szűrőalgoritmust is. Egyszerűen állítsuk a Table komponens Filtering tulajdonságát True-ra. Ekkor a rendszer minden egyes rekordra meghívja az OnFilterRecord eseményt. Az ehhez az eseményhez rendelt metódusban tetszésünk szerinti szűrést végezhetünk. Íme egy példa:

```
procedure TDataModule2.Table1FilterRecord(
  DataSet: TDataSet; var Accept: Boolean);
begin
  if (Table1Country.Value = 'US') or
       (Table1Country.Value = 'US Virgin Islands') or
       (Table1State.Value = 'Jamaica') then
    Accept := True
  else
    Accept := False;
end;
```

Ezt a szűrési szabályt is az adatmodulhoz kötöttük, ezért mindkét nézetet befolyásolni fogja. Az adatmodul továbbfejlesztésénél a fentihez hasonló, fix szabályok mellett megengedhetjük a felhasználónak, hogy saját szabályokat is megadjon, például a korábbi tartományválasztó párbeszédablakkal. Pontosan ezt tettem a TwoView3 példában (amelyben a fenti kód is szerepel, de csak megjegyzésként).

Az új Choose Range (tartományválasztás) párbeszédablakban egy újabb jelölőnégyzet és két listapanel szerepel (lásd a 17.15. ábrát), amelyeket a form létrehozásakori országokkal illetve államokkal töltöttem fel:

```
procedure TFormRange.FormCreate(Sender: TObject);
begin
  with DataModule2 do
  begin
    Table1.First;
    while not Table1.EOF do
    begin  // add unique values
      if ListBoxCountries.Items.IndexOf (
           Table1Country.AsString) < 0 then
        ListBoxCountries.Items.Add
             (Table1Country.AsString);
      if ListBoxStates.Items.IndexOf
             (Table1State.AsString) < 0 then
        ListBoxStates.Items.Add (Table1State.AsString);
      Table1.Next;
    end;
    Table1.First;
  end;
```



17.15. ábra *A TwoView3 példa új párbeszédablaka, a továbbfejlesztett szűrési lehetőségekkel*

Amint a program feltöltötte a listapaneleket, azok rögtön megjelennek a tartomány-opciókkal együtt. Az adatmodul ChooseRange metódusának ismét meg kell jelenítenie a párbeszédablakot, ellenőriznie kell a tartományt, be kell állítania a szűrőket és frissítenie kell a táblát. Ez azért szükséges, mert ha a táblaszűrés már aktivizálva volt de a szabályok megváltoznak, akkor a Delphi nem számolja újra automatikusan az aktív rekordok listáját. Lássuk ennek a metódusnak az újabb verzióját:

```
procedure TDataModule2.ChooseRange;
begin
  with FormRange do
    if ShowModal = mrOK then
    begin
      if CheckBoxRange.Checked then
        Table1.SetRange ([Edit1.Text], [Edit2.Text])
      else
```

```
            Table1.CancelRange;
          Table1.Filtered := CheckBoxFiltering.Checked;
          Table1.Refresh;
        end;
    end;
```

Ennek a kódnak a legfontosabb része az OnFilterRecord események kezelője, amely most már ellenőrzi, hogy az aktuális ország vagy állam ki van-e jelölve valamelyik listapanelben (amelyek megengedik a többszörös kijelölést, de a tartománykijelölést nem). Íme a kód:

```
procedure TDataModule2.Table1FilterRecord(
  DataSet: TDataSet; var Accept: Boolean);
begin
  {if the item corresponding to the country in the
  listbox is active, then view the record}
  with FormRange.ListBoxCountries do
    if Selected [Items.IndexOf
        (Table1Country.AsString)] then
      Accept := True
    else
      Accept := False;
  with FormRange.ListBoxStates do
    if Selected [Items.IndexOf (Table1State.AsString)]
then
      Accept := True;
end;
```

Figyeljük meg, hogy a második részben az Accept értékét hozzá kell OR-olni az első részbelihez. Valójában egyszerűen beállíthatom True-ra a korábbi értéktől függetlenül (hiszen a „vagy True" mindig True), vagy meghagyhatom korábbi értékét (mivel a „vagy False" mindig a korábbi értéket tartja meg).

Egyéni szűrés és a kliens/szerver fejlesztés

Ha az előző példa kódját nézegetjük, láthatjuk, hogy a program nagyon sokat kommunikál az adatbázissal. Metódusok böngésznek a táblában az értékeket vizsgálva, a legnagyobb értéket illetve adott stringeket keresve. Sőt, van még egy Object Pascal-ban megírt szűrő kód is. Mint azt láthattuk, ennek a kódnak a legnagyobb része nincs közvetlen kapcsolatban a felhasználói felülettel, hanem logikailag az elválasztott adatmodulhoz tartozik. Ez látszólag helyes háromrétegű modell, de lehet rossz kliens/szerver megközelítés is.

Gyakorlatilag a kliensre további számításokat bíztunk azáltal, hogy az adatmodulba ágyaztuk a kódot, míg a szervernek nem sok munkát adtunk (mivel egy Paradox táblát használtunk, ez logikus volt, de most az általános esetről próbálok meg beszélni). Ha a tábla nagyobb lenne és egy SQL szerveren lenne elhelyezve, akkor programunkat egyáltalán nem mondhatnánk hatékonynak.

Ilyen esetekben továbbra is használhatjuk az adatmodult, de meg kell próbálnunk több lekérdezést használni (amiket a szerver hajt végre) és ezekre kicserélni az adatfeldolgozó metódusainkat (amiket a kliens hajt végre).

A TwoView3 példánál maradva használhatnánk lekérdezéseket, amelyek megadják a legnagyobb vevőkódot, a különböző állam és országneveket illetve egy bonyolult `where` kifejezéssel szűrhetnénk a rekordokat és megadhatnánk egy tartományt. Ezt az új változatot nem fogom itt megírni, de elgondolkodhatunk egy jobb háromrétegű megközelítésen, ahol a középső réteg számításigényességét minél inkább a szerverre terheljük. Ezen megközelítés következő lépése egy vagy több tárolt eljárás megírása lehetne.

Összegezve a Delphi 3 képes megkülönböztetni a 3 réteget (adatnézet, közbülső szabályok és a valódi adatelérés) valamint nagy segítséget nyújt az ilyen alkalmazások a kliens (Object Pascal kód), a középső réteg (egy adatmodul Object Pascal kódjának szerveren történő futtatása) és az adatbázis szerver (SQL kód) közötti felosztására. Hogy hogyan alkalmazzuk ezt a stratégiát az rajtunk és a tapasztaltságunkon múlik, ugyanis nem mindig könnyű egyszerre megtalálni a legjobb felépítésű megoldást miközben a lehető legnagyobb sebességet is szem előtt tartjuk.

Az adatszótár

Gyakorta használunk hasonló elrendezéssel rendelkező (például megegyező kijelzési formájú) mezőket egy programon belül, vagy akár több különböző alkalmazásban. Ha egész vagy decimális számokat, százalékos értékeket, telefon vagy faxszámokat (esetleg ugyanazon számokat csak különböző előtagokkal) vagy hasonló szabványos mezőket használunk, akkor nagyon fárasztó munka egyenként, ismételten beállítanunk őket. Ezért tartalmaz a Delphi adatszótárt (Data Dictionary). Ez egy olyan adatbázis, amely mezők tulajdonságait tárolja. A szótár segítségével megadhatjuk a gyakran használt mezők tulajdonságait, vagy egyszerűen átmásolhatjuk azokat már meglévő mezőkről (természetesen minden további nélkül használhatjuk a már meglevő adatszótárbeli elemeket is).

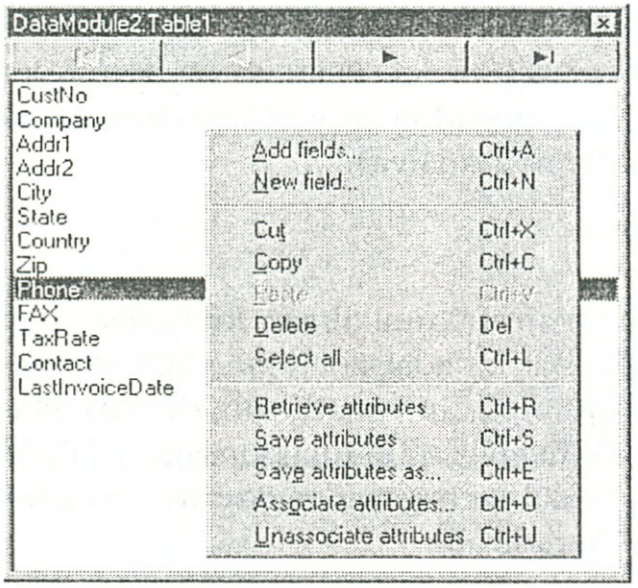
Egy – esetleg több programozós – kliens/szerver környezetben az adatszótár az adatok további megosztása érdekében lehet akár egy távoli szerveren is.

Megjegyzés

Bár az alapértelmezett adatszótárat egy Paradox táblával ábrázolják, újat akár egy SQL szerveren elhelyezett táblaként is létrehozhatunk.

Az adatszótár és a mezőszerkesztő (Fields Editor)

Az adatszótárt érintő legtöbb művelet egy tábla vagy lekérdezés komponens mezőszerkesztőjében meg végbe. A mezőszerkesztő helyi menüjében tulajdonképpen öt adatszótárral kapcsolatos parancs található (17.16. ábra).



17.16. ábra *A mezőszerkesztő helyi menüje, az adatszótár kezelésére vonatkozó parancsokkal*

Íme egy lista arról, mit is csinálnak ezek (az eredetinél logikusabb sorrendben):

- *Associate Attributes (Attribútum társítás)*: ezzel a paranccsal lehet egy attribútumhalmazt egy mezőre alkalmazni. Először ki kell jelölnünk egy attribútumhalmazt, amit az aktuális mezőre alkalmazni az akarunk. Amikor egy mezőre alkalmazunk egy attribútumhalmazt, akkor az tulajdonságok értékei valójában csak bemásolódnak a mező tulajdonságaiba.
- *Unassociate Attributes*: ez a parancs az ellentéte az attribútum társításnak, törli a mező és az attribútumhalmaz kapcsolatát.
- *Retrieve Attributes (Attribútumok begyűjtése)*: ezzel a paranccsal lehet lekérni a kapcsolt attribútumhalmaz értékeit. Csak akkor használható, ha a mező társítva van az attribútumhalmazzal. Egy betöltő parancsként lehet leginkább elképzelni.
- *Save Attributes (Attribútumok elmentése)*: ez a parancs a Retrieve Attributes parancs ellentéte. Elmenti az aktuális mező tulajdonságainak értékeit a társított attribútumhalmazba. Ha ilyen társítás nincs, akkor bekér tőlünk egy új nevet, ahogyan a következő, Save Attribute As parancs teszi.

- *Save Attributes As*: ez a parancs való arra, hogy társítsunk egy mezőt egy új attribútumhalmazzal, amihez a megjelenő ablakban egy nevet kell megadnunk.

E parancsok eléggé maguktól értetődőek, az adatszótár rövid használata után megtanulhatjuk, hogyan is kell használni őket. Elég gyorsan hozzá fogunk szokni használatukhoz. A mezők és attribútumok közötti társítások használatával nagyon egyszerűen tudjuk a megfelelő tulajdonságokat egy új, valamely már meglevővel megegyező nevű mezőre alkalmazni. Például minden „telefon" mező automatikusan társításra kerül egy megadott attribútumhalmazzal (a már a szótárban levő USPhones bejegyzést felhasználva).

Miből áll egy attribútumhalmaz?

Az előző részben többször használtam az attribútumhalmaz kifejezést. Egy attribútumhalmaz egy bejegyzés (egy rekord) az adatszótárban. Egy attribútumhalmaz egy TField objektum több tulajdonságára vonatkozik, de más általános tulajdonságokat is leír. Egy attribútumhalmaz több tulajdonsága a TField egyes leszármazottaira vonatkozik; ezek általában nagyon egyszerűen megérthetők. Következzék egy lista ezen attribútumokról:

`Alignment`	`DisplayWidth`	`ReadOnly`
`BlobType`	`EditFormat`	`Required`
`Currency`	`EditMask`	`Transliterate`
`DisplayFormat`	`MaxValue`	`Visible`
`DisplayLabel`	`MinValue`	
`DisplayValues`	`Precision`	

Természetesen ha egy olyan tulajdonságot módosítunk, mint például a Precision (pontosság; csak lebegőpontos számokra értelmezett), azután ezt az attribútumhalmazt egy TStringField mezőre alkalmazzuk, akkor ez az érték figyelmen kívül hagyódik.

Az attribútumhalmaz néhány más tulajdonsága a mező általános viselkedésére vonatkozik és azt határozza meg, hogyan kell egy új mezőobjektumot elkészíteni egy táblában vagy lekérdezésben.

Attribútum	Használat
TField Class	Megadja a mező típusát (a TField egy alosztályát), amit akkor kell használni, amikor egy új mezőt adunk az adathalmazhoz.
TControll Class	Megadja, milyen típusú lesz az az adatfüggő komponens, amit a Delphi akkor készít el, amikor egy mezőt a mezőszerkesztőből a formra húzunk. Ha itt nincs semmilyen érték megadva, akkor a Delphi a hagyományos megközelítés szerint a mezőobjektum típusát használja.
Based On	Ezt az értéket akkor kell megadnunk, amikor egy attribútumhalmazt új néven mentünk el. Megadja, melyik attribútumhalmazra épül a mostani. Ezt azt jelenti, hogy ha változtatunk az eredeti halmaz egy elemén és ebben a halmazban azt nem definiáltuk felül, akkor a változások az aktuális halmazban is érvényesek lesznek. Nekem erről a szövegszerkesztők stíluslapjaival való analógia jut eszembe, ahol is a stílusok egymásra épülnek. Természetesen egyfajta öröklődésre is hasonlít.

Az adatszótár használata

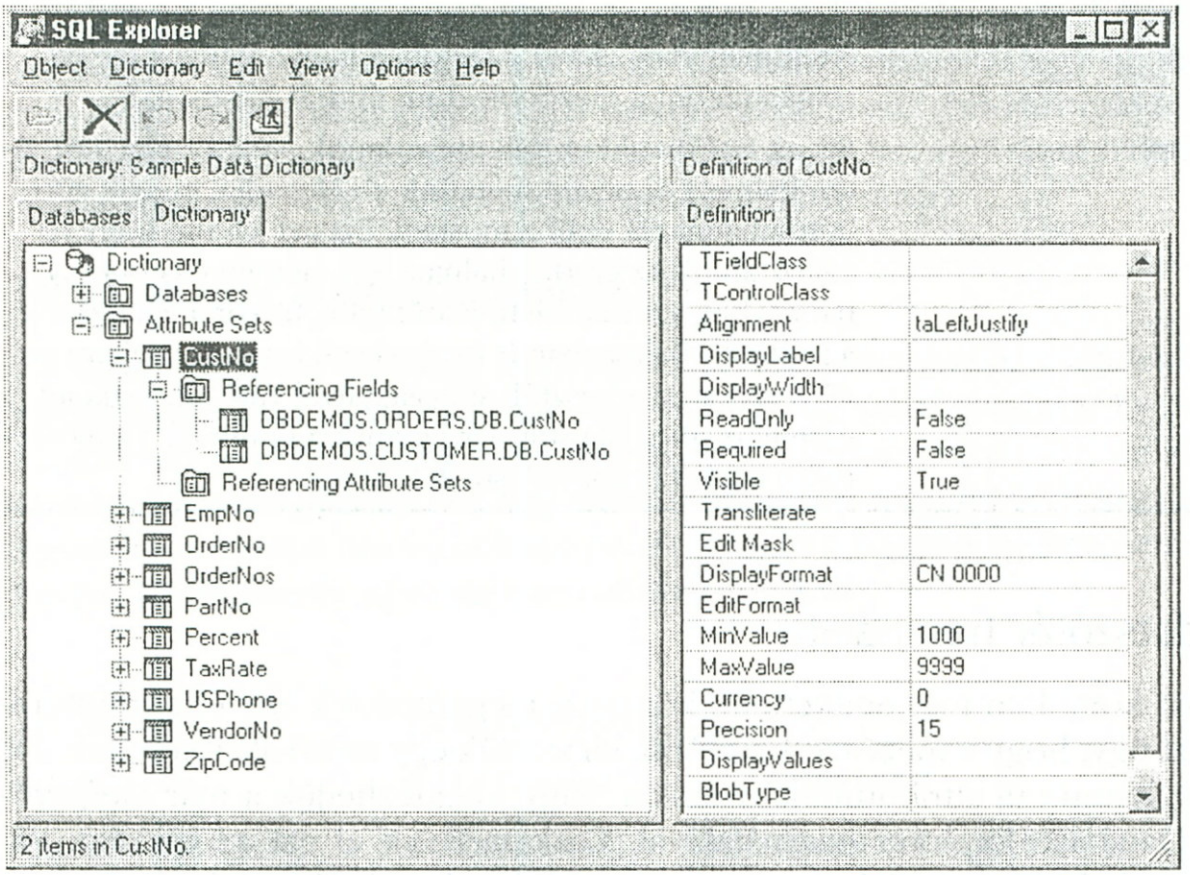
Mint azt korábban már említettem, könnyen megadhatunk egy új attribútumhalmazt úgy, hogy a mezőszerkesztőből elmentjük egy mező attribútumait. De létrehozhatunk új attribútumhalmazokat, vagy megnézhetjük a már meglévőket a Database Explorer használatával. Csak indítsuk el ezt az eszközt, válasszuk a bal oldalon a **Dictionary** (szótár) fület, és már láthatjuk is a Default Data Dictionary-t (DefaultDD, alapértelmezett adatszótár), amely a BDESDD Paradox táblára épül. Ha már mi is létrehoztunk adatszótárakat, akkor itt azokat is megtalálhatjuk. A **Dictionary** bejegyzés alatt két fát találunk, a Databases-t és az Attribute Sets-t, amint azt a 17.17. ábra mutatja.

Az Attribute Sets alatt megtalálhatjuk az attribútumhalmazokat, mindegyiket a tulajdonságaik értékeivel együtt. Szintén megtalálhatjuk itt az ezekre alapuló adatbázis táblák és más attribútumhalmazok listáját. A Database Explorerben akár módosíthatjuk is ezeket az attribútumhalmazokat vagy újakat hozhatunk létre.

Az adatbázis táblák mezői és az attribútumhalmazok közötti társításokat is megtekinthetjük és módosíthatjuk is a második, Databases nevű fa tanulmányozásával. Amikor kiválasztjuk egy tábla egy mezőjét, egy kombinált lista használatával társíthatjuk azt egy az adatszótárban már meglevő attribútumhalmazzal.

Egészét tekintve ha egy új projektet kezdünk és már megterveztük az adatbázis-táblákat, véleményem szerint érdemes létrehoznunk az

attribútumhalmazokat és azok társításait, mielőtt valóban dolgozni kezdenénk a Delphiben. Ha viszont még nincs teljesen kidolgozva a tervünk, akkor nyugodtan használjuk a mezőszerkesztőt az adatszótár és a táblák felépítéséhez, majd később használjuk az SQL Explorert a helyzet áttekintéséhez és dokumentálásához.



17.17 ábra *Az adatszótár, amint azt a Database Explorerből nézzük*

Tranzakciókezelés

Ha SQL szerverrel dolgozunk (illetve akkor is, ha helyi adatbázisfájlokkal dolgozunk) alkalmazásaink biztonságosabbá tételéhez használhatunk tranzakciókezelést. Egy tranzakció tekinthető úgy, mintha egy sor műveletet egyetlen, „atomi" egészként kezelnénk. Az atomiság alatt most tulajdonképpen oszthatatlanságot értünk. Egy példa talán sokat segíthet fogalom tisztázásában. Tegyük fel, hogy egy cégnél minden alkalmazott bérét egy rögzített összeggel emelni kell, csakúgy mint az előző fejezetbeli Total példában. Ha ezen művelet közben valamilyen hiba történik, akkor lehet, hogy vissza akarjuk vonni az addigi változtatásokat. Ha most az „emeljük fel minden alkalmazott bérét" műveletet egyetlen tranzakciónak képzeljük el, akkor az vagy teljes egészében megtörténik, vagy teljesen figyelmen kívül marad.

Az adatbázisműveletek egyetlen tranzakcióként való elvégzésének hasznos előnyei vannak: elkezdhetünk egy tranzakciót, elvégezhetünk több, valójában egynek tekintett műveletet, majd a végén elvégezhetjük a módosításokat, vagy visszagörgethetjük a tranzakciót az összes eddigi művelet eldobásával. Általában ez utóbbit akkor szokás megtenni, ha valamilyen hiba történt a műveletek valamelyikénél.

A tranzakciókezelés Delphiben nagyon egyszerűen megvalósítható. Alapértelmezés szerint minden szerkesztő műveletet egyetlen tranzakciónak tekintünk. Ha ezen változtatni akarunk, akkor azt explicit módon jeleznünk kell. Egyszerűen helyezzünk egy Database komponenst egy formra vagy adatmodulba, kapcsoljunk minden táblát illetve lekérdezést erre a formon vagy adatmodulban található komponensre (mindjárt megnézzük hogyan), majd használjuk a TDatabase osztály alábbi három metódusát:

`StartTransaction`	Elkezd egy tranzakciót.
`Commit`	Jóváhagyja a tranzakció megkezdése óta megtett módosításokat.
`Rollback`	Visszaállítja az adatbázis tranzakció megkezdése előtti állapotát.

Egy egyszerű példa a tranzakciókezelésre

A tranzakciókezelés példájához ahelyett, hogy egy bonyolult, tranzakciókezelést igénylő helyzetet állítanék össze, inkább megoldottam, hogy a felhasználó maga tudja kezelni a három műveletet. Nem ez a tranzakciókezelés szokásos módja, de segít a működés megértésében.

A Transact példa formja egy panelre helyezett három gombból, valamint a maradék területet elfoglaló rácsból áll. A rácsot egy egyszerű lekérdezéshez kapcsoltam, amely egy adatbáziskomponenshez van kötve. Egyszerűen helyezzük az adatbáziskomponenst a formra, adjunk értéket a DatabaseName tulajdonságának (ami nem ugyanaz, mint a Name tulajdonság, mivel ez utóbbi csak a komponens neve), majd adjuk értékül ezt az adatbázisnevet a Query komponens DatabaseName tulajdonságának. Összefoglalva a form fő komponenseinek értékei az alábbiak, a szokásos szöveges leírással:

```
object Form1: TForm1
  Caption = 'Transact'
  object DBGrid1: TDBGrid
    Align = alClient
    DataSource = DataSource1
  end
  object Panel1: TPanel
```

```
    Align = alTop
    object BtnStart: TButton...
    object BtnCommit: TButton...
    object BtnRollback: TButton...
  end
  object Database1: TDatabase
    AliasName = 'IBLOCAL'
    Connected = True
    DatabaseName = 'MyData'
    Params.Strings = (
      'USER NAME=SYSDBA')
    SessionName = 'Default'
    OnLogin = Database1Login
  end
  object DataSource1: TDataSource
    DataSet = Query1
  end
  object Query1: TQuery
    Active = True
    BeforeEdit = Query1BeforeEdit
    DatabaseName = 'MyData'
    RequestLive = True
    SQL.Strings = (
      'select * from Employee')
  end
end
```

Most már megnézhetjük a példa kódját is, amely igen egyszerű. Amikor megnyomjuk az első gombot a program új tranzakciót kezd valamint engedélyezi a másik két gombot:

```
procedure TForm1.BtnStartClick(Sender: TObject);
begin
  Database1.StartTransaction;
  BtnStart.Enabled := False;
  BtnCommit.Enabled := True;
  BtnRollback.Enabled := True;
end;
```

A Commit gomb meghívja az adatbázisobjektum megfelelő metódusát, a Database1.Commit-ot, ezután megfelelő módon engedélyezi illetve letiltja a többi gombot. Az utolsó gomb, a Rollback, szintén fel kell frissítse a DBGrid tartalmát. Ennek megtételéhez elég lezárni majd újra megnyitva végrehajtani a lekérdezést:

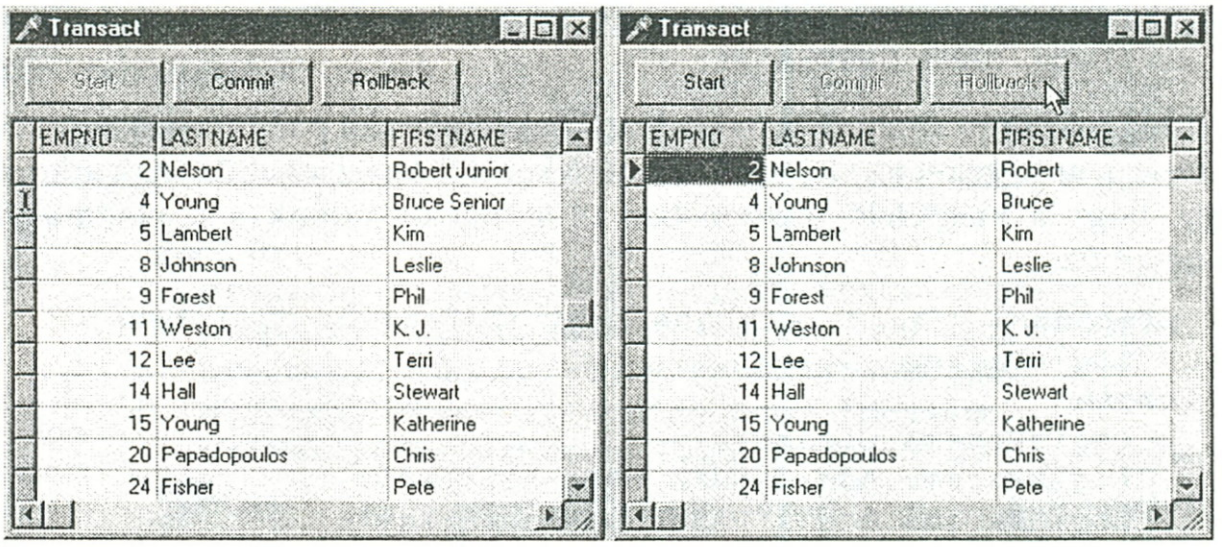
```
procedure TForm1. BtnRollbackClick(Sender: TObject);
begin
    Database1.Rollback;
    Query1.Close;
    Query1.Open;
```

```
    BtnStart.Enabled := True;
    BtnCommit.Enabled := False;
    BtnRollback.Enabled := False;
end;
```

Ha most megpróbáljuk lefuttatni a programot, tapasztalni fogjuk, hogy nyugodtan változtatgathatunk bármit az adatbázisban, a Rollback gombra kattintva minden módosítást vissza tudunk állítani. Még csak a Start gombot sem kell megnyomnunk, ha a kódját minden szerkesztési művelet előtt végrehajtjuk:

```
procedure TForm1.Query1BeforeEdit(DataSet: TDataSet);
begin
  // start a transaction, if not already started
  if not Database1.InTransaction then
    BtnStartClick (self);
end;
```

Figyeljük meg, hogy ez a kód még az előtt lefut, hogy a DataSet szerkesztő módba kerülne. A 17.18. ábrán a visszagörgetés eseményét figyelhetjük meg.



17.18. ábra *A Transact példa eredménye a Rollback művelet előtt (bal oldali ábra) és után (jobboldali ábra)*

Egyéni adatbázis-belépés

A fenti Transact példa bemutatott egy újabb (az eddigiektől független) lehetőséget, az egyéni adatbáziskapcsolatot. Amikor ezt a példát elindítjuk, a Delphi nem tárja elénk a megszokott bejelentkező párbeszédablakot. Ezt azért van, mert úgy döntöttem, lekezelem a Database komponens OnLogin eseményét.

Ebben az eseménykezelőben átadhatjuk a bejelentkezési paramétereket az adatbázis számára, esetleg így testre szabva a bejelentkezési eljárást. A

példában egyszerűen csak továbbléptem ezen, fix értékeket átadva az adatbázis bejelentkezési eljárásának.

Megjegyzés

Ha előre megadott értékeket használunk a felhasználói névnél illetve a jelszónál, akkor az adatbázis védelmét teljesen megkerüljük. Bármelyik felhasználó aki ezzel a programmal ellátott számítógépet használ, ugyanazt a nevet foga használni, jelszóellenőrzés nélkül. Helyi InterBase szerver használata esetén ez logikus lehet, de egy igazi SQL szerver esetén ez a megközelítés nagyon veszélyes.

Amint azt a fenti form szöveges leírásán is láthatjuk, az adatbázis komponensnek van egy néhány ilyen formájú paramétere:

```
Params.Strings = (
  'USER NAME=SYSDBA')
```

Ez a string már azon általunk is megadható bejelentkezési paraméterek egy részét képezi, amelyeket az OnLogin eseménykezelőben az adatbázisnak adunk át úgy, hogy a megfelelő metódusban új stringeket adunk a LoginParams paraméterhez:

```
procedure TForm1.Database1Login(Database: TDatabase;
  LoginParams: TStrings);
begin
  LoginParams.Clear;
  LoginParams.Add ('USER NAME=SYSDBA');
  LoginParams.Add ('PASSWORD=masterkey');
end;
```

Ezeknek a stringeknek speciális alakjuk van: két oldalukat egy egyenlőségjel választja el. A TString osztálynak van egy kevésbé ismert lehetősége, a Value tulajdonság, mellyel tömb formájában adhatunk értéket ezeknek a stringeknek. Ebben az esetben ez a jelölés nagyon hasznos lehet, mert átírhatjuk a kódot a következő alakra:

```
procedure TForm1.Database1Login(Database: TDatabase;
  LoginParams: TStrings);
begin
  LoginParams.Values ['USER NAME'] := 'SYSDBA';
  LoginParams.Values ['PASSWORD'] := 'masterkey';
end;
```

Ugye szép? Bár a TString-ek ezen Value tulajdonságát főleg a adatbázisparaméterek és hasonló értékek átadására használjuk, ezt a technikát felhasználva nagyon egyszerűen implementálhatunk asszociatív stringlistákat.

Adatbázishibák kezelése

Az adatbázisprogramozás lehetőségei közül ebben a fejezetben utolsóként a hibakezelés különböző lehetőségeivel foglalkozunk. Persze hagyhatnánk, hogy a Delphi egy kivételről szóló üzenetet mutasson nekünk valahányszor egy hiba történik, de gyakran inkább megpróbálnánk kijavítani azokat, vagy csak több részletet megtudni róluk.

Bár a Delphi legtöbb kivételosztálya egyszerűen csak egy hibaüzenetet hordoz, az adatbázis-kivételeknél a hibák listáját is láthatjuk, a helyi BDE kódokkal és az SQL szerver natív hibakódjaival együtt. A Message tulajdonság mellett az EDBEngineError osztálynak két másik tulajdonsága is van, az ErrorCount és az Errors. Ez utóbbi az előfordult hibák listáját tartalmazza:

```
property Errors[Index: Integer]: TDBError;
```

A lista minden egyes eleme egy TDBError típusú objektum, melynek a következő tulajdonságai vannak:

```
type
  TDBError = class
    ...
  public
    property Category: Byte read GetCategory;
    property ErrorCode: DBIResult read FErrorCode;
    property SubCode: Byte read GetSubCode;
    property Message: string read FMessage;
    property NativeError: Longint read FNativeError;
  end;
```

Ezt az információt felhasználva írtam egy egyszerű adatbáziskezelő programot, amely a hibákat egy feljegyzés komponensben mutatja meg. A hibák lekezelésére ez a DBError példa a globális Application objektum OnException eseményét használja. Ezt a műveletet a form OnCreate eseménykezelőjében valósítottam meg:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnException := MyError;
end;
```

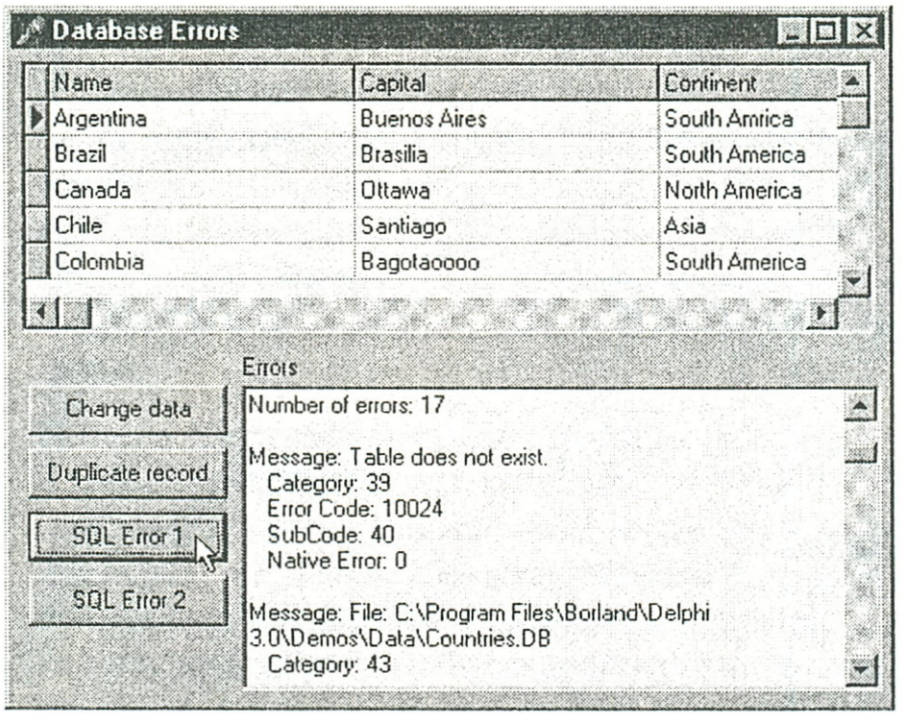
A MyError metódus csak meghívja az adatbázishiba részleteit megmutató metódust, ha az egy EDBEngineError:

```
procedure TForm1.MyError (Sender: TObject; E:
Exception);
begin
  Beep;
  if E is EDBEngineError then
    ShowError (EDBEngineError (E))
  else
    ShowMessage (E.Message);
end;
```

Azért választottam le a hibát megjelenítő kódot, hogy megkönnyítsem a dolgunkat, ha esetleg egy másik környezetben is fel szeretnénk használni. Itt van a ShowError metódus kódja, amely minden fellelhető információt megjelenít a formra helyezett Memo1 komponensben:

```
procedure TForm1.ShowError(E: EDBEngineError);
var
  I: Integer;
begin
  Memo1.Lines.Clear;
  Memo1.Lines.Add('Error: ' + (E.Message));
  Memo1.Lines.Add('');
  Memo1.Lines.Add('Number of errors: ' +
    IntToStr(E.ErrorCount));
  Memo1.Lines.Add('');
  // iterate through the Errors records
  for I := 0 to E.ErrorCount - 1 do
  begin
    Memo1.Lines.Add('Message: ' +
      E.Errors[I].Message);
    Memo1.Lines.Add('    Category: ' +
      IntToStr(E.Errors[I].Category));
     Memo1.Lines.Add('    Error Code: ' +
      IntToStr(E.Errors[I].ErrorCode));
    Memo1.Lines.Add('    SubCode: ' +
      IntToStr(E.Errors[I].SubCode));
    Memo1.Lines.Add('    Native Error: ' +
      IntToStr(E.Errors[I].NativeError));
    Memo1.Lines.Add('');
  end;
end;
```

A hibakezelő kódon kívül ebben a példában szerepel egy tábla, egy lekérdezés és egy, a táblához kapcsolt DBGrid. Ezt felhasználva végezzünk valamilyen illegális műveletet, például vegyünk fel egy rekordot olyan kulccsal, ami már szerepel a táblában. Ilyen és hasonló hibákat tudunk okozni a feljegyzés komponenstől balra levő négy gombbal, amint azt a 17.19. ábra mutatja.



17.19. ábra *A DBError példa harmadik gombjára kattintva 17 adatbázishibát kapunk!*

Az első gomb egy olyan kivételt okoz, melyet nem az adatbáziskezelő rész vált ki, így nem a mi hibakezelőnk jelenik meg, hanem egy szokásos üzenetablak:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table1.FieldByName ('Name').Value := 'something';
end;
```

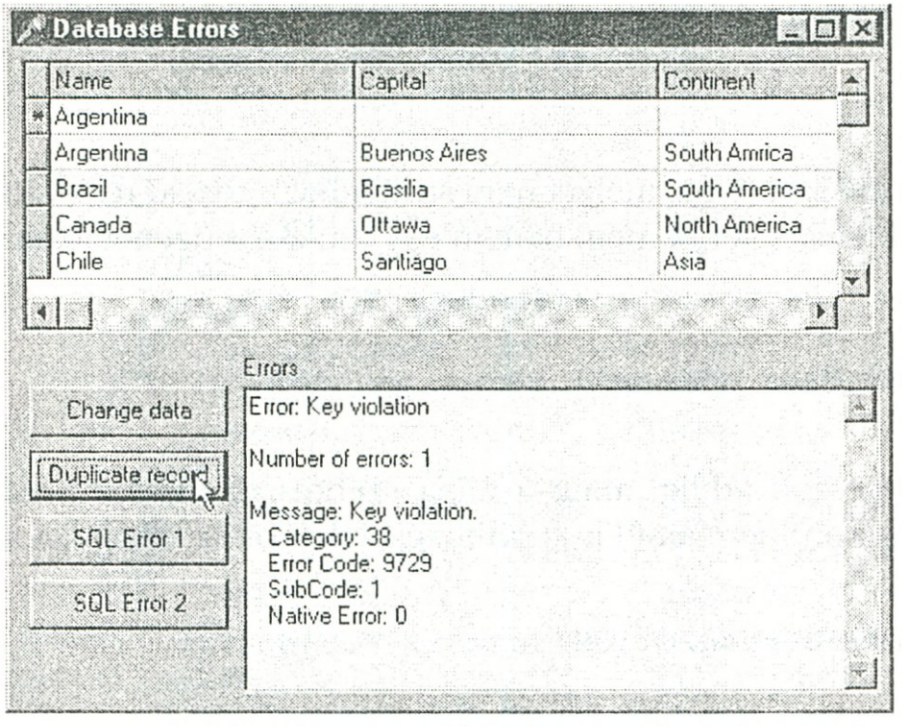
Ez a kód érvénytelen egészen addig, amíg a tábla szerkesztő módba nem kerül. A második gomb elkerülhetetlenül kulcsütközési hibát okoz, ahogyan a 17.20. ábrán látjuk:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  S: String;
begin
  s := Table1.FieldByName ('Name').Value;
  Table1.Insert;
  Table1.FieldByName ('Name').Value := s;
  Table1.Post;
end;
```

Az utolsó két gomb a Query komponensre vonatkozik; mindkettő egy-egy érvénytelen SQL lekérdezést hajt végre. Az első esetben a program egy nem létező táblával próbál meg dolgozni (nézzük meg ismét a 17.19. ábrát az

eredményről), míg a második esetben egy nem létező rekordot akar használni. A kész metódus kódja:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  Query1.SQL.Clear;
  Query1.SQL.Add (
    'select * from Countries where Population>100000');
  Query1.Open;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  Query1.SQL.Clear;
  Query1.SQL.Add (
    'select * from Country where Populations>100000');
  Query1.Open;
end;
```



17.20. ábra *A DBError példa második gombjára kattintás egy egyszerű kulcsütközési hibát okoz*

Hogyan tovább?

A kliens/szerver programozás már elég régen nagyon közkedvelt téma. Az olyan fogalmak, mint „upsizing", „downsizing" vagy „rightsizing" egyre általánosabbá válnak (akármit is jelentsenek; ezek a fogalmak különböző emberek számára különböző dolgokat jelentenek). Ha ezek voltak a tegnap jelszavai, akkor a máé biztosan a háromrétegű modell. De ismétlem, az emberek ezt a fogalmat is arra használják, amit az Ő rendszerük csinál, ahogyan azt megszokhattuk.

Ebben a fejezetben csak néhány dolgot néztünk meg, a problémákat csak felületesen, néhány egyszerű, de hasznos példán keresztül megvizsgálva. A célom egyszerűen csak az volt, hogy az Olvasót kíváncsivá tegyem a kliens/szerver alkalmazásfejlesztés terén. A programozásnak ebbe a területébe már nagyon sok munkát – és pénzt – beleöltek, így a személyi számítógépek és a nagyobb gépek között olyan híd alakult ki, ami korábban elképzelhetetlen lett volna. A legtöbb általam említett téma valójában mélyebb tanulmányozást érdemel, ami viszont messze túlmutat jelen Delphi programozási könyv keretein.

Ezzel a fejezettel a könyv adatbáziskezeléssel foglalkozó része lezárul, bár később – a 28. fejezetben – még foglalkozunk egy adatbázis-közeli eszközzel, a jelentéskészítéssel.

A következő fejezetben a komponensfejlesztéssel és az ezzel kapcsolatos témákkal foglalkozunk majd. Figyelmünk egyre bonyolultabb dolgok felé irányul majd, pedig a fejezet még csak a legelső lesz a harmadik, haladó részben.

Az első két részben tisztáztuk az alapokat és foglalkoztunk a komponensek használatával. Mostantól beleássuk magunkat a komponensek építésébe és az OLE valamint COM technikákkal való munkába. Ezután – még előrébb tekintve – a negyedik részben még több időt töltünk majd el a haladó Delphi programozási témák között.

Ha kihagytuk (részben vagy egészben) a 6. és 7. fejezeteket, akkor ahogyan a kezdőknek ajánlottam, itt az ideje visszatekinteni és újra átolvasni őket, még mielőtt a továbbiakba belekezdenénk.

Delphi 3 Mesteri Szinten a piacon elérhető Delphi könyvek közül a legjobb kritikát kapta. A könyv tartalmát teljes mértékben a Delphi 3-hoz igazították, így az olvasó egy minden tekintetben jól használható könyvet vehet a kezébe. A Delphi programozás egyik elismert mestere, Marco Cantù a legérthetőbb és legátfogóbb művet készítette el e témáról.

A Delphi 3 Mesteri Szinten igazi értéket ad mind a kezdő, mind a haladó programozók kezébe:
– A 32 bites Delphi programozás exkluzív bemutatása
– Több száz mintaprogram tanulmányozása, felhasználása
– A 32 bites kontrolok, a párhuzamos programozás, az OLE és a Registry használata és sok egyéb érdekes téma.

Minden OOP lehetőséget felfedezhet a Delphi 3-ban.
Megtanulhatja az osztályok, objektumok, unitok, metódusok és interfészek használatát az Object Pascal nyelv segítségével. A könyv megmutatja, hogyan lehet használni a vizuális form leszármaztatást, az Object Repository-t és további új, hatékony eszközöket.

Mesteri szintre vezeti az olvasót a Windows 95 és a kliens/szerver programozás művészetében.
Bemutatja az Interneten is használható ActiveX és ActiveForm komponenseket; bemutatja, hogyan használhatjuk a form stílusokat, kereteket, felbontás, méretezést és elhelyezést; bemutatja, hogyan készítsünk felhasználóközelibb programot grafikai elemekkel és a TChart komponenssel; bemutatja, hogyan készítsünk eszközsorokat, CoolBar-okat, státuszsorokat; és bevezet minket a menük használatának mélyebb szintjeire, mint például a tulajdonos által rajzolt menük készítése.

Hogyan készítsünk saját komponenst?
A könyv bemutatja, hogyan készítsünk komplex, az igényeinknek megfelelő komponenseket, és hogyan fejlesszünk tovább már létezőket. Részletezi az „packages" használatát, amivel könnyebben tudjuk használni és forgalmazni a komponenseket.

Adatbázis-kezelő alkalmazások készítése Windows 95 alá
Megismertet minket a Varázslók, adatmodulok, multirecord objektumok, lock-up fájlok, a DBGrid, és a QuickReport komponensek használatával, hogy még egyszerűbben és még hatékonyabban készíthessünk adatbázis-kezelő alkalmazásokat.

Marco Cantù könyvíró, tanár, tanácsadó. A *Mastering Delphi*, a *Delphi Developer's Handbook* és a *Borland C++ 4.0 Object Oriented Programming szerzője*. Emellett több újság számára készít cikkeket, többek között a Windows Tech Journal és a The Delphi Magazine számára. Windows és Delphi programozást tanít, híres nemzetközi előadó.

Felhasználói szint:
Kezdőtől profiig
Könyv típus:
Kézikönyv/referencia
Kategória:
Windows programozás

Eladási ára: 2980 Forint
ISBN 963 03 5348 2
9 789630 353489

SYBEX®
Kiskapu Virtuális Könyvesbolt
http://www.kiskapu.hu/kiskapu/
KISKAPU
KISKAPU KIADÓ